AD-A034 390 WHARTON SCHOOL OF FINANCE AND COMMERCE PHILADELPHIA P--ETC F/G 9/2
DETERMINISTIC VERSUS NONDETERMINISTIC PROCEDURE FOR AUTOMATIC P--ETC(U)
OCT 76 D J ROOT N00014-75-C-0462
UNCLASSIFIED 76-10-01 NL

1 OF 2
AD
A034390

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 76-10-01
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): Deterministic Vs Nondeterministic Procedure for Automatic Program Generation in DBTG Data Base Access
5. TYPE OF REPORT & PERIOD COVERED:
6. PERFORMING ORG. REPORT NUMBER: 76-10-01

7. AUTHOR(s): David J. Root
8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0462

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Department of Decision Sciences
The Wharton School
University Of Pennsylvania, Phila., PA
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR 049-272

11. CONTROLLING OFFICE NAME AND ADDRESS:
Office of Naval Research
Information Systems, Arlington, VA 22217
12. REPORT DATE: Oct 76
13. NUMBER OF PAGES: 92

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
(same)
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: (does not apply)

16. DISTRIBUTION STATEMENT (of this Report)
Reporduction in whole or in part is permitted for any purpose of the United States Government.

DISTRIBUTION STATEMENT A
Approved for public release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
(same)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
DBTG
Network Data Base
Query Language
Automated Programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
As information systems grow in scope and size, costs associated with the programming activity are quickly becoming major factors in the economic feasibility of such systems. One obvious solution is to enlist the computer itself to aid in the programming activity. The form of such aid could range from interpreters to program synthesizers (automatic program generation). To date few program synthesizers have been used in

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

"real world" applications due to either need for more powerful AI techniques to solve the problems involved, or to the costs of the existing AI techniques which they employ.

This paper describes the work involved in minimizing the use of AI in one such program synthesizer, the Automatic Program Generator (APG), in its application to report generation from network (DBTG) data bases.

DETERMINISTIC VS NONDETERMINISTIC PROCEDURE FOR AUTOMATIC PROGRAM GENERATION IN DBTG DATA BASE ACCESS

by

David J. Root

76-10-01

Prepared for the
Office of Naval Research
Information Systems
Arlington, Va. 22217
under
Contract N0014-75-C-0462
Project No. 049-272

Department of Decision Sciences
The Wharton School
University of Pennsylvania
Philadelphia, Pennsylvania

ACKNOWLEDGEMENT

I thank Professor Rob Gerritsen, my thesis advisor, for the direction and advice with which he has provided me. Portions of Sections 2., 3., 4. and 7. of this paper have been taken, with appropriate changes, from his PhD thesis [Gerritsen 1975]. I would also like to thank my wife, Christine, for encouragement as well as for much of the typing.

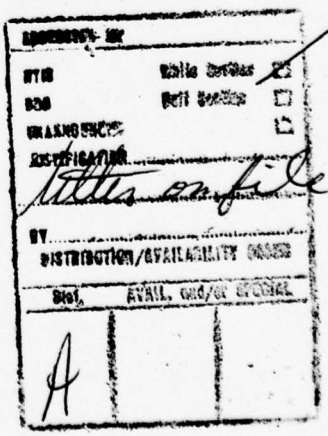


Table of Contents

Table of Contents (cont'd)

1.0 INTRODUCTION

As Information Systems grow in scope and size, the time and cost involved in the programming activity are quickly becoming major factors in the economic feasibility of such systems. The signifigance of the programming activity is also an important current topic due to the costs attributed to program errors [Baker 1973]. Efforts in the area of "automated programming" or "automatic program synthesis" may hold some solutions for these problems.

In the past, higher level languages (then considered automated programming) helped to gain greater efficiencies in the programming activity. Today, automated programming, or automatic program synthesis, covers a somewhat nebulous area. As explained by Siklossy and Sykes [1975]:

> "Although a compiler for a high level language might be considered a synthesizer, since it transforms an algorithm written in the language into executable machine code, generally discussions of program synthesizers are restricted to those systems which transform into code descriptions which are "far" from being executable. The concept of "far" is relative to the state of knowledge."

The system which will be described in this paper falls into the gray area of this definition. Merely presented with the input and output of the system, the reader would probably conclude that the system qualifies as an automatic program synthesizer, as defined above. However, once the reader becomes aware of the degree to which it was possible to mechanize this task, he may feel that the system falls in an area closer to the case of the high level language compiler.

Hoare [1972] identifies three aspects of a computer program that determine its success as regards its task:

1. That those aspects of the real world with which the program is concerned are completely and correctly represented in the logic used to develop the program.
2. That the behavior of the program coincides with the steps called for by the logic.
3. That the representation of the real world and the method of manipulating that representation are such as to result in acceptable program running costs.

Therefore the problem of automated programming is twofold. First, the knowledge which is used by programmers to meet these criteria must be identified and formalized (e.g. structured programming and the Logic of Programs). Second, there must be a method to give this knowledge to the computer and have the computer utilize it efficiently in generating programs.

Buchanan [1974] has developed an Automated Programming Generator (APG), which utilizes the Logic of Programs [Hoare 1972, Hoare and Wirth 1972], primitive functions and procedures, axioms, definitions and rules of program composition to generate programs to accomplish given goals. This system is reviewed more extensively in Section 2. Gerritsen [1975] applied the APG, with some modification, in developing a system for use in the generation of reports from network type (DBTG) data bases. If the reader does not have a previous knowledge of network data bases, it would probably be helpful to first read something about them. Gerritsen [1975] provides very complete coverage

of the concepts involved in structuring of and data retrieval from network (DBTG) data bases, or the basics of DBTG data bases are covered in Date [1975].

Buchanan's APG System and Gerritsen's application of it to DBTG data base access accomplish the program generation through a non-deterministic procedure implemented using backtracking and pattern invoked procedures as provided by Micro-Planner [Hewitt 1971, Sussman and McDermott 1972]. Several others, Siklossy and Sykes [1975], Green and Barstow [1975], Manna and Waldinger [1975], and Haseman and Whinston [1975], to mention a few, are developing automatic program synthesizers which differ from Buchanan's APG on points such as the use of recursive structubes rather than iterative loops or producing program traces prior to code generation, but all heavily utilize the backtracking capability of Micro-Planner or some similar system to give their system "intelligence". This paper explores the benefits of making the process as deterministic as possible. The system first carries out any necessary search (planning) and then executes a deterministic procedure to generate the program code.

It is hoped that the work discussed in this paper, along with the planned extensions, will provide a better understanding of the general logic which underlies a specific class of programs, that is the class of programs used to extract information from network data bases. Knowledge of this logic should give insight into the planning capabilities which program synthesizers (automated or human) will need in order to function economically in solving complex tasks.

2.0 REVIEW OF THE APG SYSTEM

Program construction is carried out using a domain independent automatic program generation system, hereafter denoted by APG, reported in [Buchanan and Luckham 1974; Buchanan 1974]. To sketch the logical basis of the APG, some elements of the logic of programs are reviewed. Also the formalism for describing APG (called a Frame) and its use in program generation are illustrated. Sections 2.1 and 2.2 have been condensed directly from the original reports [Buchanan and Luckham 1974; Buchanan 1974; Igarashi, London and Luckham 1973; Hoare 1969].

2.1 Logic Of Programs

Statements of the logic are of the form P{A}Q where P,Q are Boolean expressions (often called assertions) and A is a program or program part. P{A}Q means "if P is true of the input state and A halts then Q is true of the output state".

A rule of inference is a transformation rule from the conjunction of a set of statements (premises, say H1 ,...,Hn) to a statement (conclusion, say K). Such rules are denoted by

$$\frac{H1,\ldots,Hn}{K}$$

2.2 Frames

The rules in a frame F are of three kinds:

PROCEDURES transform states into states and are expressed as statements in the logic of programs.

SCHEMES are methods for constructing programs and are expressed as rules of inference in the logic of programs.

RELATIONAL LAWS: definitions and axioms which hold in all states and serve to "complete" incomplete state descriptions by permitting first order deduction of other elements of a state from those given.

A problem for program construction may be stated as a pair <I,G>, where I is an input assertion (or initial state) and G is the output assertion (or goal that must be true in the output state). The program construction task is to construct a program A such that I{A}I`, where I`$\supset$G. A solution is the sequence of rules of F used in the construction of the solution program A.

Notation:

Substitutions, denoted by α, do not replace any variable that occurs in the initial state I. Expressions, all of whose variables occur in the initial state are called "fully instantiated". |- denotes a first order deduction using F and the standard rules described below.

Standard rules: A set of rules representing standard programming knowledge are implemented in the program construction methods of the problem solving algorithm:

R0. Assignment Axiom: $P(t)\{x \leftarrow t\}P(x)$

R1. Rule of Consequence:
$$\frac{P \supset Q, Q\{A\}R}{P\{A\}R} \qquad \frac{P\{A\}Q, Q \supset R}{P\{A\}R}$$

R2. Rule of Composition:
$$\frac{P\{A\}Q, Q\{B\}R}{P\{A;B\}R}$$

R3. Rule of Invariance: if $P\{A\}Q$ and $I \vdash P$ then $I\{A\}Q$ I' where I' is the largest subset of I consistent with Q.

R4. Change of Variables:
$$\frac{P(x)\{A(x)\}Q(x)}{P(y)\{A(y)\}Q(y)}$$

R5. Conditional Rule:
$$\frac{P \wedge Q\{A\}R, \quad P \wedge \tilde{}Q\{B\}R}{P\{\textit{IF } Q \textit{ THEN } A \textit{ ELSE } B\}R}$$

Frame rules: A Frame defines a programming environment using the rules described below. These rules are used in conjunction with the standard rules to generate programs.

S1. Primitive procedures (or operators): the rule defining procedure r is of the form $P\{r\}Q$. The assertions P and Q are the pre- and post-conditions of r. r must contain a procedure name and parameter list.

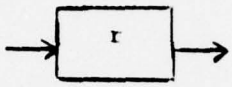
S2. Iterative rules: an iterative rule definition containing the Boolean expressions P(basis), Q(loop invariant), R(iteration step goal), L(control test) and G(rule goal) is a rule of inference of the form:

$$\frac{P, \vdash Q, \quad Q \wedge L\{?\}R, \quad R\{??\}Q \vee \tilde{}L, \quad \tilde{}L \supset G}{P\{\text{while L do ?;.??}\}G}$$

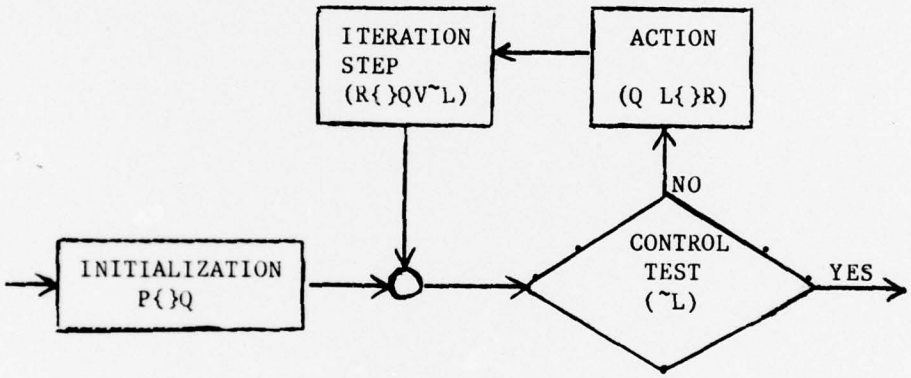
S3. Definitions: A definition of G in terms of P is a logical equivalence |- P≡G.

S4. Axioms: A frame axiom P is a logical axiom |- P.

S1 type rules generate a single line of code. The result is a module consisting of a single operation:



S2 type rules generate code for an iteration loop as represented in the module:



Where INITIALIZATION and ITERATION STEP are generated by a specified combination of S1, S2 and S3 type rules, and ACTION is generated by some combination of S1, S2 and S3 type rules.

S3 type rules construct modules from modules (submodules) produced by specified S1, S2 and S3 type rules, through composition or alternation.

2.3 Program Generation

The theorems are used in a recursive subgoaling procedure to generate information retrieval programs. The recursive procedure first builds a plan for the target program in depth-first fashion. The plan is a tree and the branches from a node correspond to the subgoals spawned at that node. For example, if the current goal is R, and R is not directly true in the current state, then the system examines the set of theorems and selects one which has a post-condition, say Q, that matches the current goal, i.e. $R \in Q\alpha$ for some substitution α. $Q\alpha$ may not be a fully bound formula, but a complete binding will be constructed during the generation process.

If the rule instance $P\alpha\{A\}Q\alpha$ achieves R as above, then $P\alpha$ becomes the current goal. If $I\{B\}P\alpha\alpha'$ (I is the current state), then by the rule of composition $I\{B;A\}Q\alpha\alpha'$, and by the rule of consequence, $I\{B;A\}R$. The system finds B;A as the program to achieve R from the initial state I.

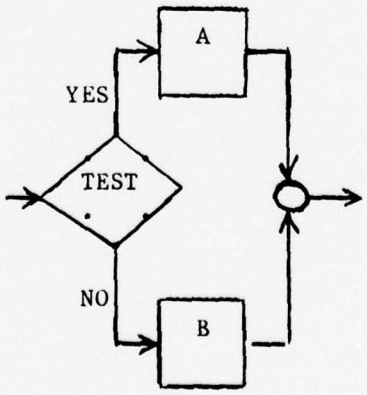
The subgoaling process does not usually distinguish, except as noted below, between the types (S1 through S4) of Frame rules. The result is that all rules are scanned for a post-condition matching the current goal. The subgoaling process does distinguish between rules of type S4 and other rules in that only S4 rules or the set of assertions can be used to prove the pre-conditions of an S4 rule. Rules of type S3 and S4 are distinguished from the other rules in that they cannot

change the set of assertions. This is a consistent distinction. Since only program segments can effect changes in the environment (when executed); only rules describing their effect should be allowed to change the state description.

Rules may be specified having a pre-condition which matches an assertion in its post-condition. Such a rule may be recursive. If it is not recursive then it may not be used to satisfy its own pre-condition.

With the APG, Buchanan also introduces an interesting improvement to the logic of programs by introducing uncertainty. The value of an assertion can be TRUE, FALSE or UNCERTAIN. This uncertain logic recognizes that there exist assertions which can only be meaningfully tested during execution of the generated program. Use of a rule containing an uncertain assertion in its pre-condition will result in the generation of a conditional procedure.

Therefore use of such a rule produces the module



Where TEST represents code which tests for the truth of the partial precondition in question. A is the module which the rule having that precondition generates assuming that the partial precondition is true. B is the module which results from invoking the last rule which was fully instantiated when called with the current state of the world, but assuming the partial precondition to be false.

The Frame is compiled by the APG to form the DMLP. Each rule in the Frame results in a Micro-Planner theorem. Such Micro-Planner constructs are not actually theorems, but that terminology will be adhered to because of historical precedent.

Each compiled theorem contains premises (or a pre-condition) and conclusions (or a post-condition). For example, the Micro-Planner theorem corresponding to the rule P{A}R has a theorem body for the pre-condition P and a calling pattern for the post-condition R.

Assertions describe the pre- and post- conditions of the rules. Assertions also describe the current state.

3.0 THE HI-IQ QUERY LANGUAGE

3.1 Introduction

Gerritsen [1975] developed the HIerarchical Interactive Query language for the specification of hierarchical reports.

Because of the preponderance of hierarchies in report structure, statistical calculation, and logical quantification, it seemed only natural to give HI-IQ a hierarchical structure. The hierarchical query structure is reinforced visually for the user by further indentations of system prompts for every hierarchical level referenced.

A HI-IQ query contains one or more hierarchical levels. Each level is used to specify a matrix in the output report, to specify the calculation of a statistic, or to check the truth value of a quantified condition. A simple one level query results in a report consisting of a single matrix which contains no statistics.

It is not unusual for the definition of a particular level to be interrupted by the definitions of lower levels. If the user desires the calculation of a particular statistic, say an average, then the system next asks him to define the calculation of that average before proceeding with the further specification of the level in which the average was requested. The prompt indentation indicates to the user which hierarchical level he is currently in.

Figure 3-1 presents a BNF description of HI-IQ. Since HI-IQ is an interactive language, only portions of a query are typed by the user. To distinguish such entries from the characters typed by the system, all system typed characters have been underlined.

Backus-Naur Form (BNF) [Naur et al 1960] is a formalism invented for the description of programming languages, specifically the grammatical structure (syntax rather than semantics) of those programming languages. BNF can also be used to describe non-programming languages, such as a restricted subset of English. An excellent description and illustration of BNF can be found in [McKeeman et al 1970].

The sequence of prompts for a particular hierarchical level always consists of three subsequences. These three subsequences are <Record name>, <CONDITION LINES> and <ITEM LINES>.

```
<QUERY>            ::=   <LEVEL>

<LEVEL>            ::=   PRIMARY RECORD FOR (<COMMAND>)
                         * <Record name>
                         CONDITIONS FOR RETRIEVAL
                         <CONDTION LINES>
                         ITEMS OR STATS <MODIFIER>
                         <ITEM LINES>

<CONDITION LINES>        ::= * NIL |
                         <CONDITION LINE> <CONDITION LINES>

<CONDITION LINE>   ::=   *OR | *ALL <LEVEL> |
                         *ANY <LEVEL> | *<TEST>

<TEST>             ::=   (<IOC> <REL> <IOCR>) |
                         (<IOC> <REL> <STAT>) <LEVEL> |
                         (<STAT> <REL> <IOCR>) <LEVEL> |
                         (<STAT> <REL> <STAT>) <LEVEL> <LEVEL>

<IOCR>             ::=   RUNTIME | <IOC>

<IOC>              ::=   <Item name> | <Constant>

<REL>              ::=   LE | LT | GE | GT | EQ | NE

<ITEM LINES>       ::=   *NIL |
                         <ITEM LINE> <ITEM LINES>

<ITEM LINE>        ::=   *<IOC> | *<STAT> <LEVEL> |
                         *REPEAT <LEVEL> | *ONE <LEVEL> |
                         *COND <TEST>

<COMMAND>          ::=   MAIN | ALL | ANY |
                         REPEAT | ONE | <STAT>

<STAT>             ::=   COUNT | TOT | AVE |
                         MIN | MAX

<MODIFIER>         ::=   TO BE DISPLAYED | FOR AVE |
                         FOR TOT | FOR MIN | FOR MAX
```

Figure 3-1. BNF for the HI-IQ language
(System prompts are underlined.)

3.2 <Record Name> Subsequence

The <Record name> subsequence consists of a single system prompt and user reply wherein the user must name the context record for the current hierarchical level. It is possible to have the system determine the context record for a particular level from the other two prompt subsequences. This would further reduce the knowledge that the user must have to use the system, but it would also increase the possibility of undetected errors because of the loss of redundancy.

3.3 <CONDITION LINES> Subsequence

The <CONDITION LINES> sub-sequence of prompts in a query level is of indefinite length. This prompt sequence defines the condition that must be true to retrieve the context record. The condition is specified using the logical connectives AND and OR and a set of tests in a disjunctive form. That is to say, if A, B, C, and D are all tests, A∧BVC∧D is equivalent to (A∧B)V(C∧D). However, the latter specification is not allowed; the user cannot control the bindings of the logical connectives, AND and OR. This is not a major restriction. Any condition can be specified in disjunctive form, albeit in a cumbersome way. Because of the immediate binding of AND, it is the default connector and need not be specified by the user.

Particular tests are of the form (A REL B). REL can have one of six values: EQ, NE, LT, LE, GT, GE. "A" can be an item name, a statistic, or a numeric or non-numeric literal. "B" can be any of these;. in addition "B" can be the keyword RUNTIME. The use of RUNTIME signals that the generated program will be an interactive program. If execution of the generated program becomes dependent on an actual value for "B", it (the generated program) will prompt the user with "A REL?", and the user`s reply will be used to determine the truth value of (A REL B).

Universal or existential quantification can be specified as part of a condition. Since quantification is only meaningful over a set of possible values, the user must be ready to define a new hierarchical level for every quantifier specified. After encountering either of the quantifiers ALL or ANY, the system automatically proceeds to prompting for the definition of a new hierarchical level.

The system also proceeds to a new hierarchical level whenever the user specifies a statistic so that the calculation of the statistic can be defined. A statistic is specified with one of the following keywords: COUNT, TOT, AVE, MIN, and MAX.

```
CONDITIONS FOR RETRIEVAL
*(PATNO EQ RUNTIME)
*(PATAGE LT 25)
*NIL

CONDITIONS FOR RETRIEVAL
*(SALARY LT 6000)
*OR
*(SALARY LT 10000)
*ANY
    PRIMARY RECORD (ANY)
    *DEPENDENT
    CONDITIONS FOR RETRIEVAL
    *(AGE LT 21)
    *NIL
*NIL
```

Figure 3-2. Two examples of retrieval conditions.

Figure 3-2 illustrates two retrieval conditions. The first is a simple conjunction of two tests. The second condition indicates that a record (employee) should be retrieved if the employee has a salary below $6,000, or if he has a salary below $10,000 and at least one dependent child.

3.4 <ITEM LINES> Subsequence

THE <ITEM LINES> subsequence is also of indefinite length and is used to define the matrix associated with the current hierarchical level. This matrix is either an output matrix (for the report) or a statistical matrix, depending on the command which invoked the current hierarchical level. The system calculates the statistic in each column

of a statistical matrix. In other words, every column is totalled or averaged, or the minimum or maximum is found in every column of the statistical matrix.

The reply to a prompt in the <ITEM LINES> subsequence must be any one of the statistical commands, an item name, the REPEAT command, the ONE command, the COND command or NIL. NIL terminates the subsequence.

Entering an item name or COUNT defines a column of the matrix. Entering any other statistic will define one or more columns of the matrix depending in turn on the number of columns in the matrix defined for that particular statistic.

Any of the commands (with the exception of the COND command) will cause the system to initiate a new hierarchical level, so that the user can further define the action associated with the command. The REPEAT command is used for generating hierarchical reports. Figures 3-3 and 3-4 illustrate a complete query and the report it defines.

```
PRIMARY RECORD(MAIN)
*DOCTOR
CONDITIONS FOR RETRIEVAL
*NIL
ITEMS OR STATS TO BE DISPLAYED
*DOCNAME
*DOCAGE
*SPECIALTY
*REPEAT
    PRIMARY RECORD (REPEAT)
    *PATIENT
    CONDITIONS FOR RETRIEVAL
    *(PATAGE GT 21)
    *NIL
    ITEMS OR STATS TO BE DISPLAYED
    *PATNAME
    *PATAGE
    *DIAGNOSIS
    *NIL
*NIL
```

Figure 3-3. Query (P1) for the report of Figure 3-4

```
FREDERICKS  41 G.P.
  SMITH     48 BOTULISM
  JONES     22 APPENDICITIS

BROWN  36   INTERNIST
  SMITH     48 BOTULISM

SLENDER     52 GEN SURGERY
  JONES     22 APPENDICITIS

BLUE   49   GYNECOLOGY
  WILLIAMSON 31 MISCARRIAGE
```

Figure 3-4. A hierarchical-matrix report.

The top level matrix of this report contains three columns for DOCNAME, DOCAGE and SPECIALTY. The secondary matrix, hierarchically nested in the top level matrix, also contains three columns, PATNAME,

PATAGE and DIAGNOSIS.

Note that in Figure 3-3, the prompt sequence for both levels of the report included all three sub-sequences as defined earlier. This query specifies a condition in the second level (on the retrieval of patients). This condition will not affect the retrieval of DOCTOR records or any other records not within the context of the PATIENT record. The condition (PATAGE GT 21) applies only to this particular context of the PATIENT record. The PATIENT record could have been referenced elsewhere in the query, and the condition (PATAGE GT 21) would not have applied.

The function of the ONE command is very similar to the REPEAT command except that only the first line of the matrix at the next level will be retrieved and displayed in the report. If the REPEAT command in Figure 3-3 is replaced with a ONE command, then the resulting report would resemble Figure 3-4 with the exception of the third line (which would not be included).

The user can control the appearance of particular attribute values on a particular line with the conditional output (COND) command. It is especially useful for exception reporting. Subsequent to encountering the COND command, the system responds as if a new hierarchical level had been specified, except that the first prompt sub-sequence is skipped. The first sub-sequence is not necessary because COND cannot change the record context.

There are a few other cases in which the full prompting sequence is not applicable, and other prompting sub-sequences will occasionally be suppressed. The third sub-sequence is not entered for the COUNT command because counting applies only to line occurrences of a matrix, the columns of the matrix do not affect it.

Similarly, it does not make sense to specify a matrix within the context of a condition quantifier (ALL or ANY), so again the third prompt sub-sequence is not entered by the system.

3.5 Port Selection

When the query has been completed the system tells the user which items, if any, might be used for a calculated direct access. To use calculated keys for port selection, each disjunct for the top level of the query must have one or more items meeting the qualifications listed below. For each disjunct the user is asked to select one of these items or none. If the user selects none for any one disjunct, then no calculated keys are used. Since an area search will have to be made to test for the disjunct for which no calculated key was selected, the other disjuncts can also be tested during the area search. Proper selection can reduce searches through the data base. For an item to qualify for use as a calculated key it must satisfy all of several restrictions:

a) It must be defined as a calculated key.

b) It must have been used in a test with an

equality relation.

c) The item must be contained in the context record for the top level of the query or in a record higher in the hierarchy than the context record for the top level of the query.

The need for the first two restrictions is obvious. The third restriction is temporary, it is necessary until a set of rules to govern the effiecient use of calckeys not meeting the restriction can be developed.

4.0 MODIFICATIONS OF HI-IQ AND APG

4.1 Introduction

In applying the APG System to the generation of the programs to produce reports from information in network type data bases, Gerritsen [1975] found it necessary to extend the original system. In the original system, program construction was accomplished through applying rules of the type P{A}Q, where Q contained a complete and specific description of the desired goal. Such a description of the desired hierarchical report would be long and complex. The length and complexity would cause two problems. First, rules to handle such goals would themselves be rather complex and therefore difficult to comprehend. Second, the number of different ways the goals can be broken into subgoals increases with the length of the goals. Since the breaking of the goals into subgoals is what builds the search tree, such long, complex goals could lead to very large search trees. This could be costly to system efficiency, since search is carried out by a depth first search of the tree.

To avoid these problems, Gerritsen had HI-IQ add assertions to the program environment (the initial state). Therefore, rather than posing a goal which is a complex description of the desired report, the initial goal is simply "write a program". When the program generator requires descriptions of the various aspects of the desired report it finds them in the state description.

In their paper, Sussman and McDermott [1972] discuss what they feel are the problems with Micro-Planner. Two of these are:

1) The inefficiencies of algorithms which employ backtracking or for that matter inefficiencies caused simply by maintaining the information needed to allow backtracking.

2) The lack of control over large goal invoked systems since it becomes difficult for the human to follow all the subgoals the system might create and try to prove in attempting to achieve its goal.

While these problems might exist in the APG System, the system does have some very strong benefits:

1) The use of the Logic of Programs aids in showing program correctness.

2) It provides a convenient method for expressing programming rules.

3) It allows the rules to be presented in a form that facilitates human comprehension of the logic involved.

In Gerritsen's application, the only search (planning) activity required is to determine the path through the network (i.e. how to get from the record defining one level of the query to the record defining one of that level's sublevels) and thereby verify that a user's request is logical vis-a-vis the data structure.

One of the major steps taken in the extension of Gerritsen's work that is described herein, was to expand the assertions generated by HI-IQ to include the results of that search. With this extension, the program generation phase became a deterministic procedure, and it was no longer necessary that the program composition rules be theorems (in the Micro-Planner sense) to allow backtracking. Therefore the compiler that translated the rules into Micro-Planner theorems was modified to translate them into LISP functions. (i.e. So that program generation would proceed deterministically rather than nondeterministically.).

This changeover necessitated one further modification of the system. As explained in Section 2.2, when a rule containing an uncertain assertion (i.e. one that is neither true nor false in the current state) in its pre-conditions is used, the generation of a conditional procedure will result. This conditional procedure is obtained by invoking the most recent fully instantiated subgoal (i.e. that subgoal for which all variables were defined when it was invoked). The state used in reinvoking this subgoal is the state that existed when the uncertain precondition was encountered but asserting the precondition to be false. In modifying the system to produce LISP functions rather than Micro-Planner theorems, it was no longer possible to permit unbound variables. This made it necessary to explicitly state which rule was to be invoked to generate the conditional procedure. Explicit inclusion of alternation in the programming rules should be an improvement vis-a-vis the original implementation. (Proposed plans for changing this method of generating conditional

procedures are mentioned in Section 9.2.)

The system functions in two phases. First the Request Handler (Sections 4.2 & 5.) interactively accepts the query from the user and generates a set of assertions describing that query. Then the Program Generator (Sections 4.4 & 6.) takes the assertions describing that query plus assertions describing the data base (Section 4.3) and generates a program to answer that query. See Figure 4-1.

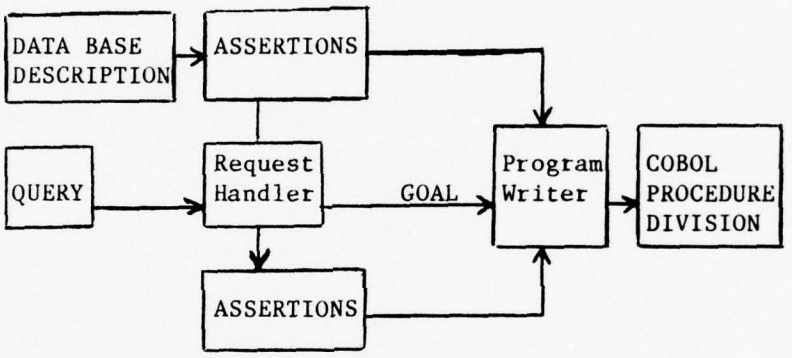


Figure 4-1. System flows.

4.2 Request Handler Assertions

The following is a description of the assertions generated by HI-IQ. These assertions are generated for a specific request and are included in the initial state description. The user specifies his request by responding to prompts from the Request Handler, and then the

Program Generator is given the task to generate a program. The possible assertions are shown in Figure 4-2.

The Request Handler describes the query with a set of assertions. An assertion must conform to one of the templates in Figure 4-2.

There is exactly one TOBEOPND assertion per query. The AREAS list indicates which areas of the data base contain records that will be accessed during the processing associated with the query. Determination of the AREAS list is not simply accomplished by tallying the names of the areas that contain the records referenced in the query. It is possible that the generated program will access areas not directly referenced via record names in the query. This situation occurs if an access path between two records passes through an intermediate record. Determination of the AREAS list therefore involves a determination of all access paths.

TOBEOPND(AREAS)
AREAS is a list of all areas containing records which may be accessed in the query.

LINKS(TYPE,MTHD,REC1,REC2,LEVEL,PATH)
LEVEL is a Dewey-decimal identification of a query level. This level has REC2 as the context record and was entered with the command indicated by TYPE from a level which had REC1 as the context record, and is reached by following PATH through the data base network. MTHD is always equal to "PATH" except if the LINKS assertion is describing the top level and a port record is being used to enter the data base, in which case MTHD is equal to "PORT".

FOR(COND,LEVEL)
COND specifies the condition whose truth must be established prior to any processing of the matrix.

TOBEUSED(ITEMS,LEVEL)
ITEMS identifies the columns in the matrix for LEVEL.

ISVAR(VAR)
VAR is a system generated variable.

Figure 4-2. Templates for Request Handler assertions.

LINKS, FOR and TOBEUSED are each asserted at most once for every level specified in the query. There is a one to one correspondence between these three and the three subsequences of prompting. The LINKS assertion defines the context of a query level and assigns the LEVEL identifier. The FOR assertion defines the retrieval condition, and the TOBEUSED assertion defines the matrix.

A Dewey-decimal scheme is used to identify the levels of the query. The top level is indentified as X. The first level occuring within the context of the top level is identified as X.1. X.2.1

identifies the first hierarchy in the second hierarchy occuring within the top level of the query.

4.2.1 LINKS Assertion -

The TYPE parameter of LINKS is actually a list containing two sub-parameters. Values of the first sub-parameter are limited to the names given in Figure 4-3. With the exception of MAIN, these are all commands which invoke new query levels. MAIN is used to identify the top level of the query and has the same interpretation as the REPEAT command.

The second TYPE sub-parameter is a unique system-created variable name. This variable is used for counting record occurences if the first sub-parameter is COUNT or AVE, or for controlling quantification if the first sub-parameter is ONE, ALL or ANY. Although TYPE will always contain a variable name as its second parameter, this variable is only used by the Program Writer if the first parameter in TYPE is one of the five commands indicated above.

The PATH parameter contains information about the path through the data base network which is followed to reach the context record for LEVEL. This parameter is discussed more fully in Section 5.2.

MAIN	ALL	TOT
REPEAT	ANY	AVE
ONE	COUNT	MIN
		MAX

Figure 4-3 possible values of the TYPE parameter in LINKS

4.2.2 FOR Assertion -

The COND parameter of FOR is a list containing all of the tests specified in the CONDITIONS FOR RETRIEVAL for a particular query level. A test is described in a sub-list containing seven entries. The first entry is the relation involved in the test and the second and fifth entries are the arguments of the test. The third entry gives the query level which defines the calculation of the first argument. If the second/fifth entry is a constant, the fourth/seventh entry is simply "CURLEV". If the second/fifth entry is a statistics command, the fourth/seventh entry is simply "NEXTLEV". If the second/fifth entry is a data item, the fourth/seventh is a path, as described in Section 5.3, which leads from the context record for the current level to the record containing the desired data item. Similarly, the sixth entry gives the level number associated with the second argument of the assertion. These level numbers will be the same as the value of LEVEL in the FOR assertion if the argument is not to be calculated but is a constant or is available from the context record.

For example, such a list might be (EQ COUNT X.1 <path1> 5 X <path2>). This test indicates that a count, as defined in the X.1 LEVEL, must be equal to 5. The path parameters are not shown here explicitly. For information on these parameters see Section 5.3.

Disjunction and conjunction is indicated in the COND list as follows. A simple list of tests represents a conjuction of those tests. A list in turn, of such conjunctions represents a disjunction. This list structure bears a close resemblance to the disjunctive form that the user must use to phrase the retrieval condition. If A, B, C and D are tests, then the COND list for A∧BVC∧D would be ((A,B)(C,D).

DMLP constructs tests to enforce quantification. If the user specifies universal quantification, the system inserts the test (ALL EQ 0). The Program Writer will eventually construct the program so that a variable associated with ALL (defined in the TYPE parameter of the LINKS assertion) is set to non-zero if the associated condition is ever false. This variable, also called a quantification flag, signals the truth value of the entire condition.

Similarly, specifying existential quantification results in a test (ANY EQ 1). The variable associated with ANY is set to non-zero in the generated procedure if the associated condition is ever true.

4.2.3 TOBEUSED Assertion -

TOBEUSED describes the matrix associated with a query LEVEL. The ITEMS parameter is again a list, each entry describing a column of the matrix. The entry describing a column is in turn also a list consisting of four entries. The first of these is an item name, statistic command, or constant. The second entry indicates the query level where the calculation of the entries in the column is defined. The fourth entry assigns a variable name which can be used by the system for the calculation of a statistic. If the first entry is a constant, the third entry is simply "NIL". If the first entry is a statistics command the third entry is simply "NEXTLEV". If the first entry is a data item, the third is a path, as described in Section 5.3, which leads from the context record for the current level to the record containing the desired data item.

As an illustration, Figure 4-4 gives the complete set of assertions derived by the Request Handler from the query of Figure 3-3. It is these assertions along with assertions describing the data base, that the Program Writer will use to generate the desired procedure.

```
TOBEOPND ((A1 A2))
LINKS ((MAIN) PATH DOCTOR X ((NIL DOCTOR AREA A1 DNHIER)))
LINKS ((REPEAT X4)
       DOCTOR
       PATIENT
       (X . 1)
       ((DOCTOR TREATMENT SET TREATING DNHIER)
        (PATIENT TREATMENT SET TREATMENTS UPHIER)))
FOR ((((GT PATAGE (X . 1) CURLEV (21) (X . 1) CURLEV))) (X . 1))
TOBEUSED (((DOCNAME X CURLEV X1)
           (DOCAGE X CURLEV X2)
           (SPECIALTY X CURLEV X3)
           (REPEAT (X . 1) NEXTLEV X10))
          X))
TOBEUSED (((PATNAME (X . 1) CURLEV X5)
           (PATAGE (X . 1) CURLEV X6)
           (DIAGNOSIS (X . 1) NIL X7))
         (X . 1)))
ISVAR (X10)
ISVAR (X7)
ISVAR (X6)
ISVAR (X5)
ISVAR (X4)
ISVAR (X3)
ISVAR (X2)
ISVAR (X1)
```

Figure 4-4 Assertions describing the query of Figure 3-3

4.3 Assertions.

A Frame consists of a set of logical statements or rules. These rules are of four different types as discussed in Section 2.2 and reviewed here:

S1 Primitive procedure rule.
S2 Iterative rule.
S3 Definition rule.
S4 Axiom.

Rules and the current state are expressed with assertions. Each assertion must correspond to a template with a semantic interpretation.

e.g. the assertion CONTAINS(PATIENT,PATNO) states that the PATIENT record contains the PATNO item.

Evaluation of an assertion determines if it is true or false in one of several ways:

(a) if previously stated to be true or false (i.e. true or false in the current state)

(b) by evaluating a rule which has the assertion in a post-condition

(c) by evaluating an ordinary LISP functions (i.e. those not output by the APG from program generation rules).

Rules may have assertions which contain variables which are to be bound to values when a match is made with an assertion of the same pattern in the data base describing the current state. Such variables are denoted in the input rules by (MATCH <variable name>). e.g. If CONTAINS(PATIENT,PATNO) is true in the current state, then evaluation of CONTAINS((MATCH RECX),PATNO) will bind RECX (a variable) to PATIENT.

DMLP uses 55 different types of assertions. Five of these were described in Figure 4-2. Figure 4-5 describes the five used to define the database structure. An earlier paper [Gerritsen 1974] illustrates the ease of conversion from a Data Definition Language specification of a data base to a set of assertions.

INAREA(RECORD,AREA)
RECORD is contained in AREA.

CONTAINS(RECORD,ITEM)
ITEM is is contained in RECORD.

DBKEY(RECORD,ITEM)
ITEM is a data base key for RECORD.

CALCKEY(RECORD,ITEM)
ITEM is a direct access attribute (calculated key) of RECORD.

HIERARCHYGROUP(RECORD1,RECORD2,SET)
RECORD1 is the owner of SET, and RECORD2 is a member of SET.

Figure 4-5. Assertions used to describe the data base.

4.4 Rule Assertions

4.4.1 Assertions For The S1 Rules. -

Figure 4-6 contains the assertions which describe the results of single program steps. These assertions occur as post-conditions of rules of type S1. Those parameters which are underlined in Figure 4-6 have a uniqueness property. For example, the system will insure that a particular ITEM will contain only one VALUE: If the assertion C(X1,0) has been made followed by a later assertion C(X1,1), then the system will erase the first assertion.

OPENED(AREAS)
AREAS is a list of areas that have been opened.

CLOSED(AREAS)
AREAS is a list of areas that have been closed.

STOPPED(NAME)
The program NAME has been stopped.

ACCEPT(VARIABLE,ITEM,RELATION)
VARIABLE contains the value entered by the user in response to the prompt "ITEM RELATION?"

CURRENT(RECORD,LEVEL)
RECORD is current at LEVEL. e.g., The named record has been found within the program segment associated with the level identifier.

INCORE(RECORD,LEVEL)
The named RECORD is in core and available for processing to the program segment associated with LEVEL.

C(ITEM,VALUE)
The named ITEM contains the given VALUE.

ANYOUTPUT(ITEMS,LEVEL)
ITEMS is a list of the columns of a matrix (that has been output at the given level)

FOUNDOWNER(RECORD1,RECORD2,SET,LEVEL)
The named RECORD1 has been found via the SET using RECORD2, a member of that SET and both records are now current at the LEVEL specified.

FOUNDNEXT(TYPE,RECORD,UNIT,LEVEL)
The next RECORD of the specified UNIT which is either an Area or Set as specified in TYPE has been found and is current for the LEVEL specified.

Figure 4-6 is continued on the next page.

FOUND(RECORD,ITEM,VALUE,LEVEL)
The named RECORD has been found using ITEM as a calculated key with the given VALUE such that RECORD is current for the specified LEVEL.

FOUNDUSING(RECORD,KEY,LEVEL)
The named RECORD has been found using the data base KEY and is current for the specified LEVEL.

FOUNDFIRST(TYPE,RECORD,UNIT,LEVEL)
The named RECORD has been found as the first record of the specified UNIT which is either a Set or Area as specified by the value of TYPE. The record is current for the specified LEVEL.

Figure 4-6. Assertions which indicate the results of single program statements.

The assertions in Figure 4-6 appear to describe the status of an executing program. The descriptions are more properly interpreted for program generation if each is read as if preceded with the phrase "Code has been generated such that...".

ISITEM(ITEM)
ITEM is contained within some record as an attribute or data base key.

BCA#(COMMAND)
COMMAND has the value "COUNT"" or "AVE".

BTMMA#(COMMAND)
COMMAND has the value "TOT", "MIN", "MAX" or "AVE"

=(A,B)
A is equal to B.

EQ#(A,B)
A is equal to B. This assertion differs from the preceeding one in that its value can be uncertain if, for example, either A or B are program variables.

TEST(CONDITION,LEVEL)
The CONDITION is true for the program segment defined for LEVEL. CONDITION is a list consisting of a relation and two arguments. When the arguments are program variables, TEST will have an uncertain value.

Figure 4-7. Other assertions.

4.4.2 Miscellaneous Assertions. -

Figure 4-7 contains a set of assertions which are difficult to classify. The first is directly derivable from the assertions describing the data base structure. The next three are used to test the values of their arguments, and the last two are used to test values or insert code to test values.

4.4.3 LISP Functions -

Some of the assertions used in the preconditions of rules are not used in post conditions of other rules or in the current state. Such assertions may be evaluated by LISP functions. Assertions for which there exists a LISP function have ## as the last two characters in their names. These assertions and their meanings are listed in Figure 4-8.

POPPATH##(PATH)
This assertion is only evaluated if MTHD in the LINKS assertion for the top level is "PORT". It pops the first item from PATH in the LINKS assertion for the top level.

POPFORLIS##(LEVEL)
This assertion pops the first item (disjunct description) from the COND list in the FOR assertion for LEVEL.

STAT##(COMMAND)
COMMAND is one of the statistical commands.

RETQ##(COMMAND)
COMMAND is a retrieval quantifier (ANY or ALL).

REPQ##(COMMAND)
COMMAND is a reporting quantifier (MAIN,REPEAT or ONE).

LITERAL##(ITEM)
ITEM is a number or non-numeric literal (enclosed in single quotes).

BINDITM##(ITEMS)
The variable ITML has been bound to a list of printable items extracted from the ITEMS list. This is done to eliminate commands which cause the printing of sub-matrices and also to replace statistic commands with the variable containing the value of the statistic.

Figure 4-8 is continued on the next page.

READL##(ITM REL)
This assertion returns the value "ITM REL?", which is used in the generated program to prompt the user.

CONTEST##(ACTION,ARG1,ARG2)
This assertion returns a test which will be used for terminating a loop. Appropriate tests for early termination of the loop depending on the ACTION of the loop are also generated. An early termination test will involve ARG1 and ARG2.

UNCERTERRSTAT##()
This assertion always evaluates to true. However it also insures that all knowledge about the value of ERRORSTATUS becomes uncertain. This is used to indicate that the value of ERRORSTATUS becomes unknown following a data base access.

Figure 4-8. Assertions evaluated by LISP.

Of the assertions in Figure 4-8, the first two are used to update the state of the world for generation of conditional procedures. The next four assertions (STAT## through LITERAL##) are very simple and return true or false depending on the value of their single parameter. The next assertion (BINDITM##) is used to bind variables to values extracted from lists. This assertion is necessary because of the list structures contained in the assertions generated by the Request Handler.

The last three assertions are unusual in that they are not evaluated for truth or failure. Instead, they return a value or change the state as is explained in Figure 4-8.

Occasionally the pre-condition of a rule will include standard LISP or Micro-Planner predicates (see [McCarthy et al 1972] and [Sussman and Winograd 1972]). These predicates are illustrated in Figure 4-9. Note that the functions CAR & CDR in Figure 4-9 can be combined to form functions. e.g. CADR(A) would be equivalent to CAR(CDR(A)).

SETQ(A,B)
Sets variable A to the value of B.

NULL(A)
A is null.

CAR(A)
Returns the first element of the list A.

CDR(A)
Returns a list equivalent to A with its first element removed.

*APPEND(A,B)
Adds list B to the end of list A.

ATOM(A)
Returns true if A is not a list but a single element.

LIST(A,B,C,...)
Constructs a list with elements A, B, C, ... Note that A, B or etc. can be atoms or lists.

SUBST(A,B,C) Substitutes A for all occurences of B in list C.

FIGURE 4-9. STANDARD LISP and Micro-Planner predicates used in the rules.

5.0 RULES FOR PLANNING

5.1 Downward Migration Of Attributes

The planning required of the system is that of "navigating" through the data base network. Before discussing this planning, it is necessary to discuss the concept of downward migration of attributes (similar to Virtual Source in the CODASYL DBTG specification [1971]). This is the concept that in a hierarchical data base structure, all attributes of a record can also be thought of as being attributes of any records which that record owns, and any records that those records own, etc. e.g. Consider the data base as illustrated in Figure 7-1. While Hospname is not actually a data element of the PATIENT record, each PATIENT record would be associated with only one Hospname value. That would be the value contained in the HOSPITAL record which owns that PATIENT record. Therefore, it is possible to think of Hospname as an attribute of a PATIENT record.

5.2 Loaction Of Context Records

Now, as to the problem of navigating through the data base network, there are two instances when the Request Handler must do this. First, there is the problem of getting to the context record for a given level. Records can be located in three different ways:

1) By a set search, or set ownership

2) By a calculated key

3) By an area search

While any of the three methods may be used in locating the top level context record(s), only the first one is used in locating all other context record(s). In Section 3.1, it was stated that all levels other than the top level are defined within the context of the higher levels. e.g. Given the data base in Figure 7-1, in the following query, the PATIENT record is referred to within the context of the HOSPITAL record.

```
PRIMARY RECORD (MAIN)
*HOSPITAL
CONDITIONS FOR RETRIEVAL
*NIL
ITEMS OR STATS TO BE DISPLAYED
*HOSPNAME
*REPEAT
    PRIMARY RECORD (REPEAT)
    *PATIENT
    CONDITIONS FOR RETRIEVAL
    *NIL
    ITEMS OR STATS TO BE DISPLAYED
    *PATNO
    *PATNAME
    *NIL
*NIL
```

It is a request for a list of all hospitals, and with each hospital a list of all its patients. The system would understand that the PATIENT records would be located by a set search of the PATSET set. A given record, Record A, can logically be referenced within the context of another record, Record B, if any of the following three conditions are satisfied.

i) B owns a set of which A is a member, or B owns a set of which C is a member and C owns a set of which A is a member, etc.

ii) A owns a set of which B is a member, or A owns a set of which C is a member and C owns a set of which B is a member, etc.

iii) A owns a set of which C is a member and B owns a set of which C is a member, A owns a set of which D is a member and D owns a set of which C is a member and B owns a set of which C is a member, etc. In this case Record C is referred to as a "common bottom".

In terms of the data base this means the HOSPITAL record could be mentioned within the context of the BILLENTRY record, or vice versa, and the PATIENT record could be mentioned within the context of the DOCTOR record, or vice versa, however, the WORKREC record cannot be mentioned within the context of the BILLENTRY record, or vice versa.

5.2.1 Set Search -

The system uses a description of the data base structure in terms of the assertions in Figure 4-2 to check for a path from the context record on a given level to the context record for each of that level's sublevels. *If no such path exists, the system informs the user with* the message: "No direct path from _____ to _____." Assuming the system finds a path, it places information describing that path in the PATH parameter of the LINKS assertion and the MTHD parameter is set equal to

"PATH", (Section 4.2.1). The PATH parameter is a list of sublists. There is one sublist for each set involved in the path. Each sublist has five elements. They are:

1. RECH: This is the name of the context record for the current record on the path.
2. RECL: This is the name of the next record on the path
3. TYP: This is always equal to "SET" for a set search.
4. NAME: This is the name of the set involved.
5. DIREC: This is equal to "DNHIER" if RECH owns the set and RECL is a member, it is equal to "UPHIER" if RECL owns the set and RECH is a member.

Therefore given the data base structure illustrated in Figure 7-1, if the PATIENT record were mentioned within the context of the DOCTOR record the path would be represented as

```
((DOCTOR,WORKREC,SET,WORKING,DNHIER)
 (WORKREC,TREATMENT,SET,TREATING,DNHIER)
 (TREATMENT,PATIENT,SET,TREATMENTS,UPHIER))
```

Rules are invoked by the system to try and prove various goals (postconditions). In the presentation of the rules postconditions are identified by the preceding >. The body of the rule is what must be true for the postcondition to be true.

In the rules (EV <expression>) is used to indicate that <expression> is not a variable or a constant but a function to be evaluated.

There are two rules which the system uses to find the path. They are:

```
UPLINK
     [=(REC1,REC2)∧
      SETQ(PATH,NIL)] V
     [HIERARCHYGROUP(REC2,REC1,ST)∧
      SETQ(PATH,(EV(LIST (LIST (REC2,REC1,SET,ST,DNHIER)))))] V
     [HIERARCHYGROUP(REC3,REC1,ST)∧
      UPLINK(REC3,REC2,PATH)∧
      SETQ(PATH,(EV(*APPEND (PATH,
                 (LIST (LIST (REC2,REC1,SET,ST,DNHIER)))))))]

     > UPLINK(REC1,REC2,PATH)

COMBOT
     UPLINK(REC3,REC1,PATH1)∧
     UPLINK(REC3,REC2,PATH2)∧
     SETQ(PATH,(EV(*APPEND (PATH2,
                    (SUBST (UPHIER,DNHIER,(REVERSE (PATH1))))))

     > COMBOT(REC1,REC2,REC3,PATH)
```

These are two of the three rules which are still implemented in Micro-Planner. Therefore the system searches all possible paths ending at the desired record, until it finds the path or exhausts all possibilities. The system will choose a path which does not use a common bottom over one which does. e.g. In the structure illustrated in Figure 7-1, the system would choose the path HOSPITAL - PATIENT to get from the HOSPITAL record to the PATIENT record over the path HOSPITAL - WORKREC - TREATMENT - PATIENT. The current implementation of HI-IQ assumes that at most one path which does not use a common bottom and at most one path which does use a common bottom exists between any two records. This is a limitation which is planned to be eliminated in future extensions to the system. It is planned to have the system generate all possible paths, and then logically eliminate as many as possible. (The basis for this elimination will be the

difficult part to implement.) If more than one path still exists, the system will then inform the user of the possibilities (along with some description of the relationship represented by the path) and ask the user to choose one.

5.2.2 Calculated Keys -

In locating the context record for the top level of the query, all three of the methods mentioned at the beginning of this section may be involved. If a calculated key is to be used, Section (3.4), the system again represents the relevant information in the PATH parameter of the LINKS assertion, but the MTHD parameter is set to "PORT". The PATH parameter is a list of sublists. There is one sublist for each disjunct in the top level of the query (as mentioned in Section 3.4, each disjunct must use a calculated key). Each sublist contains three parameters. They are:

1. CALC: The name of the data item which is the calculated key.
2. TPATH: If CALC is a data item in the context record for the top level of the query, then this parameter is set equal to that record name. If CALC is a data item in a record higher in the hierarchy than the context record for the top level of the query, then this parameter is a description of a path (Section 5.2.1) leading from the record which contains CALC to the context record for the top level of the query.
3. VALUE: This is the value which CALC was specified to have in the query.

5.2.3 System Set Search -

If a calculated key is not used to locate the top level context record then the second alternative the system will try is a system set search. This is only possible if the top level context record is a member of a system owned set, or if a record, which owns a set of which the top level context record is a member, is a member of a system set, or etc. In this case the top level context record(s) are located by a set search. The information is recorded as described in Section 5.2.1. In the first sublist RECH is set equal to "SYSTEM".

5.2.4 Area Search -

If neither of the above two methods can be used to locate the top level context record, then an area search is used. (An area search treats the area in which the record type may be located as a sequential file and an exhaustive search takes place.) The relevant information is again placed in the PATH parameter of the LINKS assertion and the MTHD parameter is set to "PATH". The PATH parameter will be as described in Section 5.2.1, however there will be only one sublist. In that sublist, the RECH parameter is set to "NIL", the RECL parameter is set to the name of the top level context record, the TYP parameter is set equal to "AREA", NAME is set equal to a list of the names of the areas to be searched, and the DIREC parameter is set equal to "DNHIER".

5.3 Location Of Data Item

The other instance in which planning is necessary is when a data item name used within the context of a given record is associated with that record through downward migration of attributes. The system places in the FOR or TOBEUSED assertions (Sections 4.2.2 and 4.2.3) the description of a path leading from the context record for the level in which the data item name is used to the record in which the data item is contained. The path is represented as described in Section 5.2.1, except that if the data item is contained in the context record for the level in which the data item is used, then instead of the list describing the path the parameter is simply set equal to "CURLEV". The system finds the path by first trying the UPLINK rule (Section 5.2.1). If that fails it tries the following rule.

```
ONPATH
     LINKS(DUM1,HREC1,REC2,LEVN,PATH1)∧
     [[ISCOMBOT(PATH1)∧
       UPLINK(CB,REC3,PATH)] ∨
      [~ATOM((EV(CAR LEVN)))∧
       ONPATH((EV(CAR LEVN)),HREC2,HREC1,PATH)]]

     > ONPATH(LEVN,REC2,REC3,PATH)
```

This is the third rule which is still implemented in Micro-Planner. ISCOMBOT is a function which returns TRUE if PATH1 has a common bottom and FALSE if it does not. An example of a common bottom is record C as described in (iii) of Section 5.2.1. (e.g. In the example (Section 5.2.1) of the path from the DOCTOR record to the PATIENT record, the TREATMENT record is a common bottom). Essentially what ONPATH does is to work backwards up the path which leads from the top level context record to the context record of the level in which the data item name

was used. For each common bottom found, it tries to UPLINK from that common bottom record to the record which contains the data item. Note that in the discussion in Section 5.1, downward migration of attributes was stated as only occurring from records containing those attributes (data items) to records owned by them, or records owned by records they own, or etc. i.e. It was stated that in a data base structure as illustrated in Figure 7-1, Docname could not in general be considered an attribute of the PATIENT record by downward migration. However, with the existence of confluent hierarcies (and common bottoms) this is not true given the proper contexts. This can be illustrated by the following query:

```
PRIMARY RECORD (MAIN)
*DOCTOR
CONDITIONS FOR RETRIEVAL
*ANY
    PRIMARY RECORD (ANY)
    *PATIENT
    CONDITIONS FOR RETRIEVAL
    *(PATNAME EQ DOCNAME)
    *NIL
ITEMS OR STATS TO BE DISPLAYED
*DOCNAME
*NIL
```

This query is requesting a list of all doctors who treat themselves. In general, given the data base structure shown in Figure 7-1, many DOCTOR records could be associated with each PATIENT record. However within the context of the above query each time a PATIENT record is obtained it is logical to the system that one and only one DOCTOR record is to be associated with it, and that is the DOCTOR record which initiated the set search which lead to that PATIENT record. It is for such occurences that the ONPATH rule is written.

The path descriptions which ONPATH and UPLINK produce lead from the record containing the data item name to the context record for the level in which the data item is used. Before placing this description of the path in the FOR or TOBEUSED assertion, the system reverses it. (i.e. It changes the path to lead from the context record to the record containing the data item.)

6.0 RULES FOR PROGRAM GENERATION

6.1 Introduction

The following is a description of the rules used in Program Generation. In the APG System, all preconditions for rules had to be stated because it was not a deterministic process and it was not possible to know anything apriori about the state that would exist when the rule was invoked. In the present case, program generation is deterministic, based on the plan described in the assertions. Therefore it is no longer necessary to state many of the preconditions for a rule since they are known to be true apriori by the fact that the process had proceeded to a point where the rule is invoked. Thus, one of the benefits of planning is that it reduces rule complexity. It is because of this that most of the S1 type rules have no preconditions.

6.2 S1 Type Rules

For the S1 type rules each is of the form P{A}R where A is a single program operation or command. Many of the preconditions, P, of the rule are not tested (stated in the rule) for reasons just explained.

Each rule in Figure 6-1 is described with A, the operation, followed in order by P, the preconditions (if any), and R, the post condition. The post condition is further identified by the preceeding >. Variables in each rule are underscored.

The operator rules are fairly simple. The interpretation of the first rule appearing in Figure 6-1 would be:

> By executing MOVE VAL TO DEST then DEST is equal to VAL (i.e. C(DEST,VAL).Note: implicit preconditions for this rule are: 1) that VAL has a value, and 2) that DEST can be assigned a value.

FIND REC RECORD is a bit more involved and would be interpreted as follows:

> If VAL is equal to RUN TIME, then generate the code to determine the desired value of ITM interactively, otherwise simply move VAL to ITM. Then by executing FIND REC RECORD, the record will have been FOUND and be CURRENT but not INCORE. Also, the value of ERRORSTATUS is uncertain, as the desired record may not exist in the database. Note: implicit preconditions to this rule are: 1) that ITM is a calculated key for REC and 2) that VAL has a value (possibly RUNTIME).

MOVE VAL TO DEST

> C(DEST,VAL)

ACCEPT VAR
ANYOUTPUT((EV(READL## ITM REL)),VAR)

> ACCEPT(VAR,ITM,REL) ∧ C(VAR,RUNTIME)

DISPLAY ITML

> ANYOUTPUT(ITM,LEVEL)

OPEN AREAS

> OPENED(AREAS)

CLOSE AREAS

> CLOSED(AREAS)

FIND REC RECORD
[~=(VAL,RUNTIME) V ACCEPT(ITM,ITM,EQ)] ∧ C(ITM,VAL)

> FOUND(REC,ITM,VAL,LEVEL) ∧ CURRENT(REC,LEVEL)
~INCORE(REC,LEVEL) ∧ UNCERTERRSTAT##()

FIND REC USING DBK

> FOUNDUSING(REC,DBK,LEVEL) ∧ CURRENT(REC,LEVEL)
~INCORE(REC,LEVEL) ∧ UNCERTERRSTAT##()

FIND FIRST REC RECORD OF UNIT TYP

> FOUNDFIRST (TYP,REC,UNIT,LEVEL) ∧ CURRENT(REC,LEVEL)
~INCORE(REC,LEVEL) ∧ UNCERTERRSTAT##()

FIND NEXT REC RECORD OF UNIT TYP

> FOUNDNEXT(TYP,REC,UNIT,LEVEL) ∧ CURRENT(REC,LEVEL)
~INCORE(OWN,LEVEL) ∧ UNCERTERRSTAT##()

GET REC

> INCORE(REC,LEVEL)

STOP

> STOP(NAM)

Figure 6-1 S1 Type Rules

6.3 S4 Type Rules

The following is a discussion of S4 type rules. These are axioms to test the truth of certain conditions in the current state. The CURAX axiom checks to see if a record is current by checking for currency at the current level in the hierarchy and if necessary, recursively checking the higher levels, within the context of which the current level was defined. This is done through use of the Dewey Decimal structure of LEVEL. (e.g. to find out if a record is current at LEVEL X.2.1, first LEVEL X.2.1 is checked; if that is unsuccessful, then LEVEL X.2 is checked).

The ITEMAX axiom checks to see if a given variable, ITM, is either a database key or a data item.

The rules are illustrated in Figure 6-2 using the convention used in presenting the S1 type rules.

CURRENT

~NULL((EV(CAR LEVEL)) ∧ CURRENT(REC,(EV (CAR LEVEL)))

> CURRENT(REC,LEVEL)

ISITEM

DBKEY(REC,ITM) ∧ CONTAINS(REC,ITM)

> ISITEM(ITM)

Fig. 6-2 S4 Type Rules

6.4 S2 And S3 Type Rules

The following is a discussion of the S2 & S3 type rules. These are the rules which actually accomplish the program composition. Each rule is given using the convention used to present the S1 and S4 type rules. Along with each rule is given an illustration of the possible program modules it may construct, and a discussion of the states (requests) which cause the rule to generate those modules. Which module is generated depends on the state at the time when the rule is invoked.

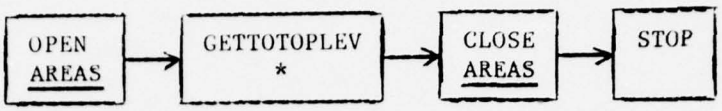
In the flowcharts, rule names in modules containing an * are expanded by the S2 or S3 Type Rule named in that module. Modules not containing an * represent single program operations.

6.4.1 PROGRAM -

This is the rule which is invoked by the system to generate the desired program after HI-IQ has added the assertions describing that program to the state description.

```
PROGRAM
      LINKS ((MATCH ACTION),(MATCH MTHD),(MATCH GREC),X,
                                              (MATCH PATH)) ∧
      TOBEOPND((MATCH AREAS) ∧ OPENED(AREAS) ∧
      GETTOTOPLEV(MTHD,ACTION,PATH) ∧
      TOBEOPND((MATCH AREAS) ∧ CLOSED(AREAS)∧
      STOPPED(NAM)

      > PROGRAM(NAM)
```

This rule results in the program module



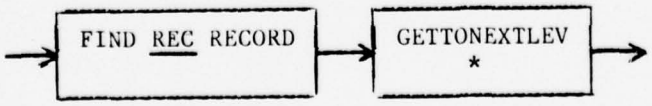
6.4.2 GETTOTOPLEV -

This rule generates code for the data retrieval and report generation.

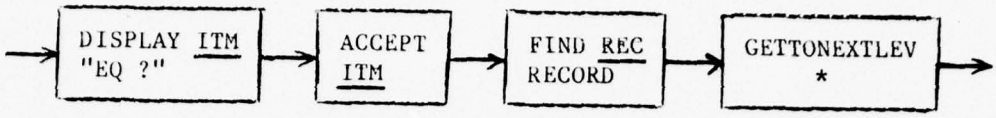
```
GETTOTOPLEV
     [=(MTHD,PATH) V
      [[ATOM((EV(CADAR PATH)))
        FOUND((EV(CADAR PATH),(EV (CAAR PATH)),
                                  (EV(CADDAR PATH)),X)
        SETQ(PATH,NIL)] V
      [FOUND((EV(CAAADAR PATH)),(EV(CAAR PATH)),
                                    (EV(CADDAR PATH)),X)
       SETQ(PATH,(EV(CADAR PATH)))]]]
     GETTONEXTLEV(ACTION,X,NIL,NIL,PATH)

     > GETTOTOPLEV(MTHD,ACTION,PATH)
```

i) If MTHD equals "PATH" (i.e. This is for a set search to evaluate or display a sublevel, or to perform a system search for location of the top level context record(s).), this rule generates the program module.

ii) If MTHD equals "PORT" (i.e. This is for the top level and data base entry is through a port record.) and the calculated key value is not to be specified interactively, this rule generates the program module



iii) If MTHD equals "PORT" and the calculated key value is to be determined interactively, this rule generates the program module

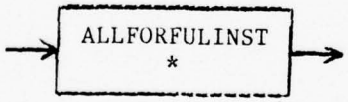


6.4.3 GETTONEXTLEV -

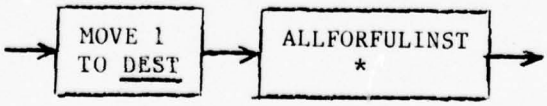
This rule generates code to move up or down hierarchies in the network and to perform any required actions.

```
GETTONEXTLEV
     [=(PATH,NIL) ∧
      [~=((EV(CAR ACTION)),ALL) V C(EV(CADR ACTION),1)] ∧
      ALLFORFULINST(ACTION,LEVEL)] V
     [=((EV(CADDR(CDDAR PATH))),UPHIER) ∧
      UPLINKED(ACTION,LEVEL,PATH)] V
     DNLINKED(ACTION,LEVEL,ARG1,ARG2,PATH)

     > GETTONEXTLEV(ACTION,LEVEL,ARG1,ARG2,PATH)
```

i) If PATH equals "NIL" (i.e. The context record for LEVEL has been made current by following PATH to its end.) and (CAR ACTION) does not equal "ALL", this rule generates the program module



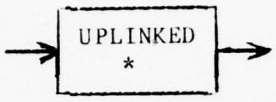
ii) If PATH equals "NIL" and (CAR ACTION) equals "ALL", this rule generates the program module



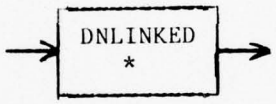
Where DEST is a flag which is set to 1 each time a record is to be

tested to see if it satisfies the conditions of the ALL command. Later code is generated to reset the flag to 0 if the record meets the conditions for the ALL command.

iii) If (CADR(CDDAR PATH)) equals "UPHIER" (i.e. The next record on the path owns a set of which the current record is a member.), this rule generates the program module



iv) If (CADDR(CDDAR PATH)) equals "DNHIER" (i.e. The next records on the path are members of a set owned by the current record.), this rule generates the program module

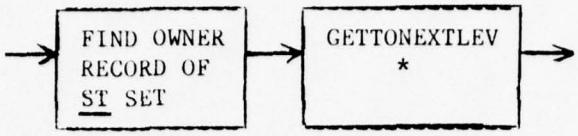


6.4.4 UPLINKED -

This rule generates code to move up a level of the hierarchy in the network.

```
UPLINKED
     FOUNDOWNER((EV(CAAR PATH)),(EV(CADAR PATH)),
                                (EV(CADDAR PATH)),LEVEL) ∧
     GETTONEXTLEV(ACTION,LEVEL,NIL,NIL,(EV(CDR PATH)))

     > UPLINKED(ACTION,LEVEL,PATH)
```

This rule generates the program module

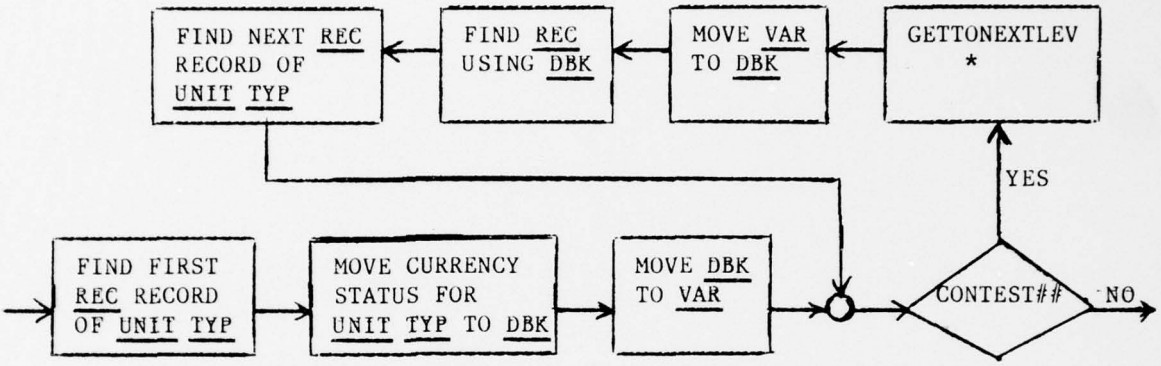


6.4.5 DNLINKED -

This is the only example of a S2 Type Rule, therefore its form is a little more complex than the other rules. As explained in Section 2.2 this rule generates a program segment of the type: While L do ?;??. The rule is defined by giving a precondition (P), a loop invariant (Q), an iteration step (R), a control test (L) and a rule goal (G). DNLINKED generates code to perform an area search or a set search.

```
DNLINKED
     precondition:
      SETQ(REC,(EV(CADDDAR PATH))) ∧
      SETQ(UNIT,(EV(CADDDAR PATH)))∧
      SETQ(TYP,(EV(CADDAR PATH))) ∧
      FOUNDFIRST(TYP,REC,UNIT,LEVEL) ∧
      DBKEY(REC,(MATCH DBK) ∧
      C(DBK,(CURSTAT<UNIT,TYP,0)

     invariant:
      C(DBK,(MATCH CURRV))

     iteration step:
      GETTONEXTLEV(ACTION,LEVEL,ARG1,ARG2,(EV(CDR PATH))) ∧
      C(DBK,(CURSTAT,A,B,CURRV)) ∧
      FOUNDUSING(REC,DBK,LEVEL) ∧
      FOUNDNEXT(TYP,REC,UNIT,LEVEL)

     loop terminator:
      CONTEST##((EV(CAR ACTION)),ARG1,ARG2)

     post condition:
      > DNLINKED(ACTION,LEVEL,ARG1,ARG2,PATH)
```

(CURSTAT,A,B,C) is a function which is expanded to "CURRENCY STATUS FOR A B". This rule generates the iteration loop. After the action of the loop has been performed, by the code generated by GETTONEXTLEV, the currency for the context record, REC, of the loop is reestablished in case it was altered during the execution of the code generated by GETTONEXTLEV.



CONTEST## sets up code for loop termination when

i) ERRORSTATUS is not equal to 0

ii) (CAR ACTION) is "ONE" and a record meeting the necessary conditions has been found and displayed.

iii) (CAR ACTION) is "ANY" and a record meeting the necessary conditions has been found.

iv) (CAR ACTION) is "ALL" and a record not meeting the necessary conditions has been found.

v) Code is being generated to evaluate "A REL B" where B is a constant, REL is EQ, NE, LT, GT, LE or GE and A is "COUNT", "MIN" OR "MAX". The value of "COUNT" or "MAX" will monotonically increase and the value of MIN will

monotonically decrease. Therefore the loop may be terminated early once the truth/falsehood of the condition "A REL B" is determined without having determined the actual value of A. (e.g. if "A REL B" is "MIN LT 10" once an instance has been found in which the value of the item of which the minimum is being calculated is LT 10 the loop can be terminated even though the true minimum may be less than the value found.)

6.4.6 ALLFORFULINST -

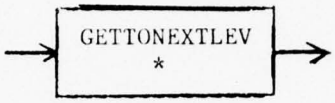
This rule generates code to perform required actions and test for any conditions on which those actions are to be based.

```
ALLFORFULINST
     [NEQ#(ERRORSTATUS,0) ∧ FOR(MATCH FORLIS),LEVEL) ∧
      ~NULL(EV(CDR FORLIS)) ∧ POPPORTS##(PATH) ∧
      POPFORLIS##(LEVEL) ∧
      GETTOTOPLEV(PORT,ACTION,LEVEL)] V
     [[[NTEST((MATCH REL),LEVEL) ∧ FOR((MATCH FORLIS),LEVEL) ∧
        SETQ(FORLIS2,(EV(CDR FORLIS)) ∧ ~NULL(FORLIS2) ∧
        POPFORLIS##(LEVEL)] V
       [FOR((MATCH FORLIS2),LEVEL) V SETQ(FORLIS2,NIL)]] ∧
      [[NULL(FORLIS2) ∧ DOACTION(ACTION,LEVEL)] V
       ALLFOR(ACTION,(EV(CAR FORLIS2)),LEVEL)]]

     > ALLFORFULINST(ACTION,LEVEL)
```

This rule appears rather complex for the modules which it generates. This complexity is due to the fact that it is the rule which is invoked to generate conditional procedures. Proposed changes (mentioned in Section 9.2) would relocate this type of activity to the rule requiring the generation of the conditional procedure, thereby making what the

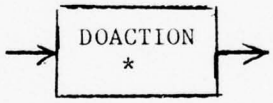
rules are doing more obvious.

i) If LEVEL equals "X" (i.e. this is the top level) and MTHD equals "PORT" and code has been generated for a previous disjunct for LEVEL and there are further disjuncts for that level, then this rule generates the program module



This would only be the case if the rule was being invoked to generate a conditional procedure. Therefore, before generating any code, the rule updates the state by executing POPPORTS## so that the next port record choosen will be for the appropriate disjunct, and POPFORLIS## so that the appropriate disjunct will be the first sublist in FORLIS of the FOR assertion for LEVEL.

ii) If there are no qualifying conditions for LEVEL, the rule generates the program module



In this case, as well as in case (iii), if the rule is being invoked to generate a conditional procedure, it executes POPFORLIS## so that the appropriate disjunct will be the first sublist in FORLIS of the FOR assertion for LEVEL.

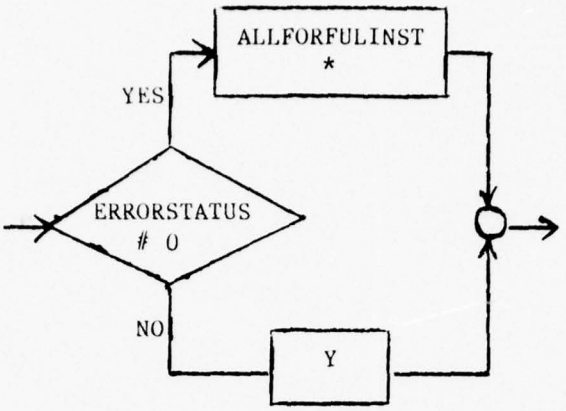
iii) If there are qualifying conditions for LEVEL the rule generates the program module

6.4.7 ALLFOR -

This rule generates code to test qualifying conditions for LEVEL and perform the associated action

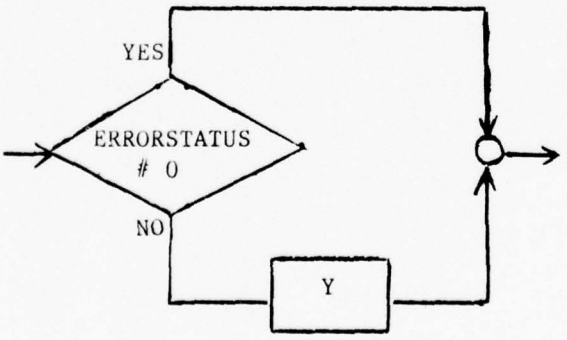
```
ALLFOR
     EQ#(ERRORSTATUS,0) ∧
     [[NULL(FORLIS) ∧ DOACTION(ACTION,LEVEL)] V
        [SETQ(ITM1,(EV(CADAR FORLIS))) ∧
         SETQ(PATH1,(EV(CADDDAR FORLIS))) ∧
         SETQ(LEVEL1,(EV(CADDAR FORLIS))) ∧
         SETQ(ITM2,(EV(CADDR(CDDAR FORLIS)))) ∧
         SETQ(PATH2,(EV(CADDDR(CDDDAR FORLIS)))) ∧
         SETQ(LEVEL2,(EV(CADDR(CDDDAR FORLIS)))) ∧
         SETQ(REL,(EV(CAAR FORLIS))) ∧
         DETVAL(ITM2,LEVEL2,PATH2,ARG2,ITM1,REL) ∧
         DETVAL(ITM1,LEVEL1,PATH1,ARG1,ARG2,REL) ∧
         TEST((EV(LIST REL ARG1 ARG2)),LEVEL) ∧
         ALLFOR(ACTION,(EV(CDR FORLIS)),LEVEL)]]

     > ALLFOR(ACTION,FORLIS,LEVEL)
```

i) If ERRORSTATUS is of uncertain value (i.e. code is being generated for the top level and data base entry is via a port record) and there is more than one disjunct left to be tested at LEVEL, this rule generates the program module



Where Y is a module as illustrated in case (7iii), (7iv) or (7v) below. This generates a conditional procedure which is executed if there was no port record in the data base which satisfy the conditions of the first disjunct in FORLIS. This conditional procedure is generated by asserting that ERRORSTATUS is not equal to 0 and invoking the ALLFORFULINST rule. That rule will generate the code to test for the next disjunct. Note: if there are no more disjuncts (case ii), then no conditional procedure is generated.

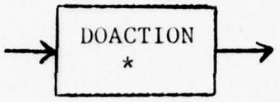
ii) If ERRORSTATUS is of uncertain value and there is just one disjunct to test for at LEVEL, this rule generates the program module



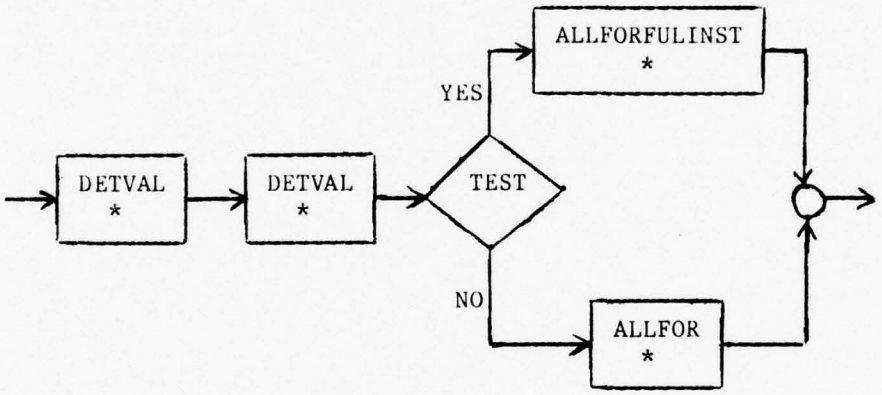
Where Y is a module illustrated as illustrated in case (7iii), (7iv) or

(7v) below.

iii) If ERRORSTATUS equals 0 and FORLIS equals "NIL" (i.e. There are no further conditions that need be tested for this ACTION.), this rule generates the program module



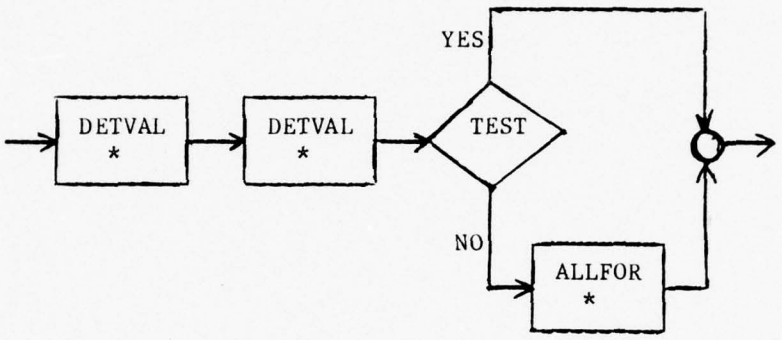
iv) If ERRORSTATUS equals 0 and FORLIS does not equal "NIL" and code is being generated for more than one disjunct, this rule generates the program module:



(Note that TEST is the negation of the condition requested, therefore if TEST succeeds the record does not satisfy the condition requested.) TEST is a partial precondition which is of uncertain value, assuming a given condition is not tested for twice in one disjunct. If a query states a given conditon twice in the same disjunct, the system will only generate code to test for it once. Therefore this case generates a conditional procedure which is executed if the record does not satisfy the condition currently being tested. This is done by invoking

ALLFORFULINST. ALLFORFULINST will check to see if code for one disjunct has been generated, and if it has, then ALLFORFULINST will begin to generate code for the next disjunct. If there are no more disjuncts (case v), then no conditional procedure is generated.

v) If ERRORSTATUS equals 0 and FORLIS does not equal "NIL" and code is being generated to test the last disjunct, this rule generates the program module



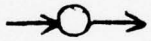
Note that two DETVAL modules appear, one for each argument of TEST (A REL B). For relations which may be used for early termination of a loop (Section 6.4.5), the statistic (MIN, MAX or COUNT) is evaluated after the value which it is being compared to.

6.4.8 DETVAL -

This rule generates code to determine required data item/statistic values.

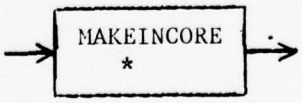
```
DETVAL
     SETQ(PARG,ITM) ∧
     [LITERAL##(ITM) V
      [=(ITM,RUNTIME) ∧ NEWLISPVAR(PARG) ∧
       GETRUNT(PARG,ARG3,REL)] V
      [ISITEM(ITM) ∧ CONTAINS((MATCH REC),ITM) ∧
       MAKEINCORE(REC,LEVEL,PATH)] V
      [[STAT##(ITM) V REPQ##(ITM) V RETQ##(ITM)] ∧
       LINKS((MATCH ACTION),(MATCH PREC),(MATCH GREC),
                                  LEVEL,(MATCH NPATH)) ∧
       [~BCA#(ITM) V C(LEV(CADR ACTION)),0] ∧
       [~RETQ##(ITM) V C((EV(CADR ACTION)),0)] ∧
       [~=(ITM,ONE) V C((EV(CADR ACTION)),0)] ∧
       [[~BTMMA#(ITM) ∧ SETQ(PARG,(EV(CADR ACTION))] V
        [TOBEUSED((MATCH ITMS),LEVEL) ∧ INITVARS(ITMS)
         SETQ(PARG,(EV(CADDAR ITMS)))]] ∧
      GETTONEXTLEV(ACTION,LEVEL,PARG,ARG3,NPATH) ∧
      [~=(ITM,AVE) ∧ DIVVARS((EV(CADR ACTION),ITMS)]] ∧
     SET(ARG,PARG)

     >DETVAL(ITM,LEVEL,PATH,ARG,ARG3,REL)
```

i) If ITM is a constant, this rule generates the program module (i.e. It generates no code.)



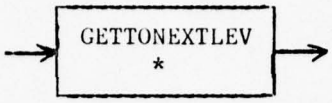
ii) If ITM equals "RUNTIME" (i.e. The item value is to be specified interactively.), this rule generates the program module

iii) If ITM is a data item, this rule generates the program module

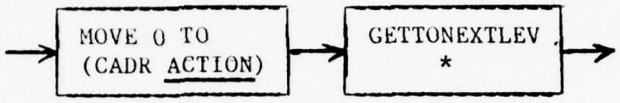


This is to bring the record containing the data item into core.

iv) If ITM equals "REPEAT" or "COND", this rule generates the program module

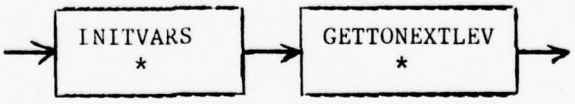


v) If ITM equals "COUNT", "ALL", "ANY", or "ONE", this rule generates the program module

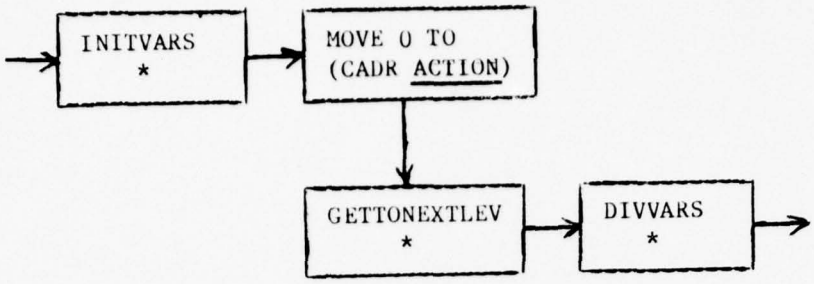


Where (CADR ACTION) is a system generated variable which is being initialized to 0. In the case of the "COUNT" command, the varible is used to store the total of the number of records found which meet the conditions for that "COUNT" command. In the cases of the "ALL", "ANY" and "ONE" commands, the variable is used as a flag. Later the system will generate code (within GETTONEXTLEV) to reset the flag to 1 to indicate that the "ANY" command has been found to be true, that the "ALL" command has been found to be false, or that the ONE command has been fulfilled.

vi) If ITM equals "TOT", "MIN", or "MAX", this rule generates the program module



vii) If ITM equals "AVE", this rule generates the program module



Where (CADR ACTION) is a system generated variable which is initialized to 0. The variable is used to keep a tally of the number of record instances over which the items being averaged are totalled.

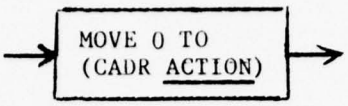
6.4.9 DOACTION -

This rule generates code to perform ACTION

```
DOACTION
     SETQ(ACTN,(EV(CAR ACTION) ∧
     [~REPQ##(ACTN) V
      [TOBEUSED((MATCH ITMS),LEVEL) ∧
       DETALLVAL(ITMS,(NIL)) ∧ BINDITML(ITMS) ∧
       [NULL(ITML) V ANYOUTPUT(ITML,LEVEL)]
       NEXTLEVOUT(ITMS)] ∧
      [~=(ACTN,ALL) V C((EV(CADR ACTION),0)] ∧
      [~=(ACTN,ANY) V C((EV(CADR ACTION),1)] ∧
      [~=(ACTN,ONE) V C((EV(CADR ACTION),1)]∧
      [~BCA#ACTN) V [SETQ(ARG2,(EV(CADR ACTION))) ∧
                     C(ARG2,(ADD1 ARG2))]] ∧
      [~BTMMA#(ACTN) V [TOBEUSED((MATCH ITMS),LEVEL) ∧
                        DETVALLVAL(ITMS,ACTION)]]

     > DOACTION(ACTION,LEVEL)
```

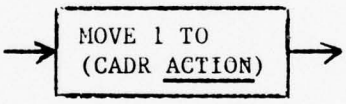
i) If ACTN equals "ONE", "REPEAT", "MAIN" or "COND", this rule generates the program module

ii) If ACTN equals "ALL", this rule generates the program module



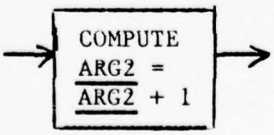
(CADR ACTION) is a flag. Before each record is tested to see if it meets the conditions for the "ALL" command, this flag is set to 1. If the record is found to meet the conditions, this code is executed to reset the flag to 0. If the flag does not get reset to 0, the control test for the iteration loop which is determining the truth value of the "ALL" command will take this as a signal that a record which does not meet the conditions for the "ALL" command has been found. Therefore the loop will be terminated, and the truth value of the ALL command is false.

iii) If ACTN equals "ANY" or "ONE", this rule generates the program module



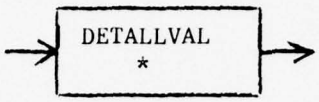
(CADR ACTION) is a flag which is set to 1 to indicate either that a record satisfying the "ANY" command was located or that the information satisfying the "ONE" command has been displayed.

iv) If ACTN equals "COUNT", this rule generates the program module

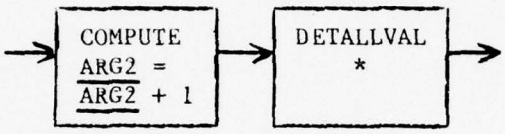


(CADR ACTION) is incremented to keep a tally of record instances for the COUNT command.

v) If ACTN equals "TOT", "MIN" or "MAX", this rule generates the program module



vi) If ACTN equals "AVE", this rule generates the program module



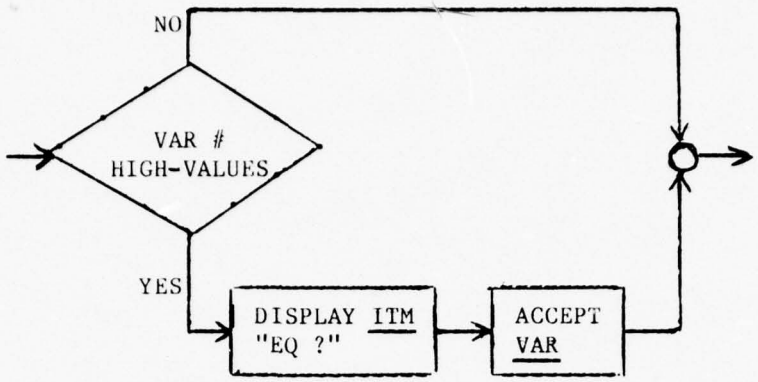
(CADR ACTION) is incremented to keep a tally of the number of items summed for the AVErage command.

6.4.10 GETRUNT -

This rule is used to generate code when a value to be specified interactively is to be used in a loop. The test is used so that the value is only requested once rather than for each iteration through the loop.

```
GETRUNT
     EQ#(VAR,HIGH VALUES) ∧
     ACCEPT(VAR,ITM,REL)

     > GETRUNT(VAR,ITM,REL)
```

This rule generates the program module



This module is used when values to be specified interactively are used within iteration loops. With this structure, the value will only be requested on the first pass through the loop, rather than on each pass through it. In a future version of the system, this request for interactive specification of the value will be brought outside of the loop and the test for prior specification of it will not be needed.

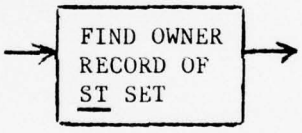
6.4.11 MAKEINCORE, FOLLOWPATH, And GETPATH -

These rules generate any code necessary to bring a desired record incore. PATH is a list of records, and relevant set names, leading from the context record for the current level to the desired record (Section 5.3). It is only possible for such a target record to be a record higher in the hierarchy than or the same record as the context record or a record made current in reaching the context record (This concept is more completely discussed in Section 5.3.). If the target record is not current then FOLLOWPATH finds the highest record in the hierarchy which is current and from there GETPATH generates code

necessary to reach the target record.

```
MAKEINCORE
     [CURRENT(REC,LEVEL) ∧
      [INCORE(REC,(MATCH LEVEL)) V GETREC(REC,LEVEL)]] V
     [FOLLOWPATH(PATH,LEVEL) ∧ GETREC(REC,LEVEL)] ∧

     > MAKEINCORE(REC,PATH,LEVEL)

FOLLOWPATH
     [~CURRENT(REC,LEVEL) ∧ GETPATH(PATH,LEVEL)] V
     FOLLOWPATH((EV(CDR PATH)),LEVEL)

     > FOLLOWPATH(PATH,LEVEL)

GETPATH
     NULL(PATH) V
     [FOUNDOWNER((EV(CAAR PATH)),(EV(CADAR PATH)),
                                 (EV(CADDDAR PATH)),LEVEL) ∧
      GETPATH((EV(CDR PATH)),LEVEL)]

     > GETPATH(PATH,LEVEL)
```

For each record in PATH which is not current these rules add the program module



Where ST SET is the set name of a member record on PATH which is current and the owner record of which is on PATH but not current. Once the target record is current, MAKEINCORE adds the program module

where REC is the target record name.

6.4.12 INITVARS -

INITVARS initializes each variable on the list ITMS to 0. These variables will be used to calculate totals, averages, maximums or minimums.

```
INITVARS
     NULL(ITMS) V
     [C((EV(CADDDAR ITMS)),0) ∧ INITVARS((EV(CDR ITMS)))]

     > INITVARS(ITMS)
```

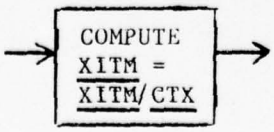
For each item, XITM, on the list, this rule generates the program module

6.4.13 DIVVARS -

TOTS contains a list of variables each of which contains the total of a value for which an average is being computed. CTX contains the count of the instances for which each TOTS variable, XITM, was summed.

```
DIVVARS
     NULL(TOTS) V
     [C((EV(CADR (CADDAR TOTS))),
                    (DIVIDE,(EV(CADR (CADDAR TOTS))),CTX)) ∧
      DIVVARS(CTX,(EV(CDR TOTS)))]

     >DIVVARS(CTX,TOTS)
```

For each variable in TOTS, this rule adds the program module

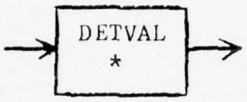


6.4.14 NEXTLEVOUT -

For each item in the list ITMS, which is a "REPEAT" or "ONE" command, NEXTLEVOUT generates the code to carry out that command.

```
NEXTLEVOUT
     [~REPQ##((EV(CAAR ITMS))) V
      DETVAL((EV(CAAR ITMS)),(EV(CADAR ITMS)),NIL,DUM,1,NIL)] ∧
     [NULL((EV(CDR ITMS)) V NEXTLEVOUT((EV(CDR ITMS)))]

     > NEXTLEVOUT(ITMS)
```

For each such item it adds the program module

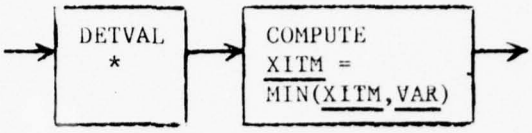


6.4.15 DETALLVAL -

For each variable in the list ITMS which is a statistic (i.e. COUNT, MIN, MAX, TOT or AVE), DETALLVAL generates the code to update variables as determined appropriate for the current context record.

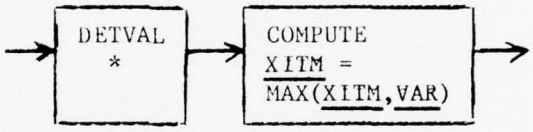
```
DETALLVAL
     REPQ##((EV(CAAR ITMS))) V
     [DETVAL((EV(CAAR ITMS)),(EV(CADAR ITMS)),
                             (EV(CADDAR ITMS)),XVAR,NIL,NIL) ∧
      [[[~=((EV(CAR ACTION)),MIN) ∧ ~=((EV(CAR ACTION)),MAX)] V
        C((EV(CADDDAR ITMS)),
          ((EV(CAR ACTION)),(EV(CADDDAR ITMS)),XVAR))] ∧
       [[~=((EV(CAR ACTION)),TOT) ∧ ~=((EV(CAR ACTION)),AVE)] V
        C((EV(CADDDAR ITMS)),
          (PLUS,(EV(CADDDAR ITMS)))]]

     > DETALLVAL(ITMS,ACTION)
```

1) For each minimum to be calculated, this rule adds the program module



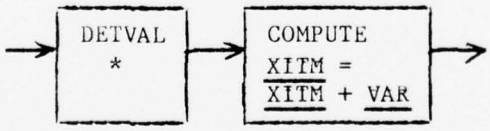
Where VAR is a system generated variable which has the value of the data item, for the current context record, for which the minimum is being calculated. XITM is used to store the minimum value of all relevant records examined so far.

ii) For each maximum to be calculated, this rule generates the proram module



Where VAR is a system generated variable which has the value of the data item, for the current context record, for which the maximum is being calculated. XITM is used to store the maximum value of all relevant records examined so far.

iii) For each AVErage or a TOTal to be calculated, this rule generates the program module



Note: the average is later calculated by DIVVARS, dividing each XITM by (CADR ACTION), the tally of the number of items summed im XITM.

7.0 EXAMPLES OF PROGRAM GENERATION

This section gives some examples of programs generated by the system from some relatively more complex HI-IQ queries. The queries are addressed to the data base illustrated in Figure 7-1.

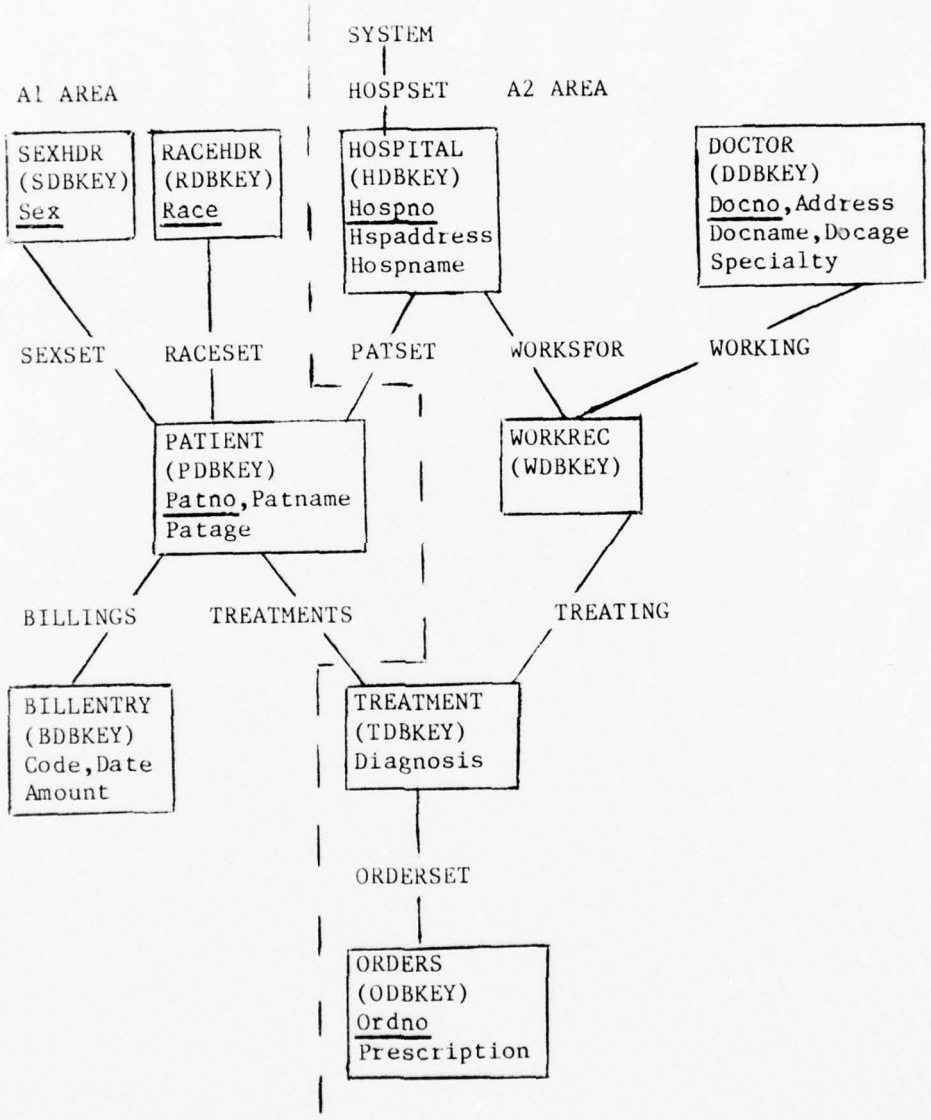


Figure 7-1. A community medical data base structure.

Program P1 (Figure 7-2) was generated from the query in Figure 3-3.

Program P2 (Figures 7-3, 7-4) will list (for a hospital specified at execution time) all patients who have accumulated a total uninsured billing of over $200. Uninsured billings are identified with one of two billing codes. This disjunction leads to the separate section called PROC3. P2 illustrates the unnecessary generation of duplicate paragraphs that may occur: PARA-105 is identical to PARA-301.

Program P3 (Figures 7-4, 7-6) generates a doctor's cross reference: for a particular doctor, each patient and all of each patient's doctors are listed. It is for programs like these, which traverse a confluent hierarchy in two directions, that conservation of the loop invariant becomes important. The loop invariant is the current of set.

The report generated by P3 will list the top level doctor in many locations: at the beginning of the report and with each patient (because he is one of the doctors associated with each of his patients). This is a minor deficiency but it can be cured with a simple extension to the DMLP. Providing for the definition of temporary variables would allow the user to differentiate between doctors at different levels of the query. This would be accomplished by specifying the storage of the doctor's number in a temporary variable at the top level of the query; e.g. SAVE DOCNO IN TDOC. Then retrieval at the third level would be specified to be conditional

on (DOCNO NE TDOC).

THE:GOAL: (PROGRAM P1):IS:ATTAINABLE:BY:THE:FOLLOWING:PROGRAM:

```
        PROCEDURE DIVISION.
        PROC1 SECTION.

        PARA-100.
            OPEN AREA A1 A2.
            FIND FIRST DOCTOR RECORD OF A1 AREA.
            MOVE CURRENCY STATUS FOR A1 AREA TO DDBKEY.
            PERFORM PARA-101 UNTIL ERRORSTATUS IS NOT EQUAL TO 0.
            CLOSE AREA A1 A2.
            STOP.

        PARA-101.
            MOVE CURRENCY STATUS FOR A1 AREA TO Z2.
            GET DOCTOR RECORD.
            DISPLAY DOCNAME DOCAGE SPECIALTY.
            FIND FIRST TREATMENT RECORD OF TREATING SET.
            MOVE CURRENCY STATUS FOR TREATING SET TO TDBKEY.
            PERFORM PARA-102 UNTIL ERRORSTATUS IS NOT EQUAL TO 0.
            MOVE Z2 TO DDBKEY.
            FIND DOCTOR USING DDBKEY.
            FIND NEXT DOCTOR RECORD OF A1 AREA.


        PARA-102.
            MOVE CURRENCY STATUS FOR TREATING SET TO Z1.
            FIND OWNER RECORD OF TREATMENTS SET.
            GET PATIENT RECORD.
            IF PATAGE IS NOT GREATER THAN 21  NEXT SENTENCE
                ELSE PERFORM PARA-103.
            MOVE Z1 TO TDBKEY.
            FIND TREATMENT USING TDBKEY.
            FIND NEXT TREATMENT RECORD OF TREATING SET.


        PARA-103.
            GET TREATMENT RECORD.
            DISPLAY PATNAME PATAGE DIAGNOSIS.
```

Figure 7-2. Program P1 in COBOL.

```
ENTER PROGRAM NAME  P2
READ DSK:?  T
   PRIMARY RECORD (MAIN)
   *HOSPITAL
   CONDITIONS FOR RETRIEVAL
   *(HOSPNO EQ RUNTIME)
   *NIL
   ITEMS OR STATS TO BE DISPLAYED
   *HOSPNAME
   *HOSPNO
   *REPEAT
      PRIMARY RECORD (REPEAT)
      *PATIENT
      CONDITIONS FOR RETRIEVAL
      *(TOT GE 200)
         PRIMARY RECORD (TOT)
         *BILLENTRY
         CONDITIONS FOR RETRIEVAL
         *(CODE EQ "X")
         *OR
         *(CODE EQ "Z")
         *NIL
         ITEMS OR STATS FOR TOT
         *AMOUNT
         *NIL
      *NIL
      ITEMS OR STATS TO BE DISPLAYED
      *PATNAME
      *PATNO
      *NIL
   *NIL
POSSIBLE PORTS ARE:
(HOSPNO)
SELECT ONE OR TYPE NIL  HOSPNO
```

Figure 7-3. Query P2: "For a hospital specified at run-time, list its name and number and the name and number of all patients whose total billings of code "X" or "Z" exceeds or equals $200."

THE:GOAL: (PROGRAM P2):IS:ATTAINABLE:BY:THE:FOLLOWING:PROGRAM:

```
        PROCEDURE DIVISION.
        PROC1 SECTION.

        PARA-100.
            OPEN AREA A1 A2.
            DISPLAY "HOSPNO" "EQ?".
            ACCEPT HOSPNO.
            FIND HOSPITAL RECORD.
            IF ERRORSTATUS IS NOT EQUAL TO 0  NEXT SENTENCE
                ELSE PERFORM PARA-101.
            CLOSE AREA A1 A2.
            STOP.

        PARA-101.
            GET HOSPITAL RECORD.
            DISPLAY HOSPNAME HOSPNO.
            FIND FIRST PATIENT RECORD OF PATSET SET.
            MOVE CURRENCY STATUS FOR PATSET SET TO PDBKEY.
            PERFORM PARA-102 UNTIL ERRORSTATUS IS NOT EQUAL TO 0.

        PARA-102.
            MOVE CURRENCY STATUS FOR PATSET SET TO Z2.
            MOVE 0 TO X5.
            FIND FIRST BILLENTRY RECORD OF BILLINGS SET.
            MOVE CURRENCY STATUS FOR BILLINGS SET TO BDBKEY.
            PERFORM PARA-103 UNTIL ERRORSTATUS IS NOT EQUAL TO 0.
            IF X5 IS LESS THAN 200  NEXT SENTENCE
                ELSE PERFORM PARA-104.
            MOVE Z2 TO PDBKEY.
            FIND PATIENT USING PDBKEY.
            FIND NEXT PATIENT RECORD OF PATSET SET.

        PARA-103.
            MOVE CURRENCY STATUS FOR BILLINGS SET TO Z1.
            GET BILLENTRY RECORD.
            IF CODE IS NOT EQUAL TO "X"  PERFORM PARA-300
                ELSE PERFORM PARA-105.
            MOVE Z1 TO BDBKEY.
            FIND BILLENTRY USING BDBKEY.
            FIND NEXT BILLENTRY RECORD OF BILLINGS SET.

        PARA-105.
            COMPUTE X5 = X5 + AMOUNT.

        PARA-104.
            GET PATIENT RECORD.
            DISPLAY PATNAME PATNO.
```

```
PROC3 SECTION.
PARA-300.
    IF CODE IS NOT EQUAL TO "Z"  NEXT SENTENCE
        ELSE PERFORM PARA-301.

PARA-301.
    COMPUTE X5 = X5 + AMOUNT.
```

Figure 7-4. Program P2.

```
ENTER PROGRAM NAME  P3
READ DSK:?  T
   PRIMARY RECORD (MAIN)
   *DOCTOR
   CONDITIONS FOR RETRIEVAL
   *(DOCNO EQ RUNTIME)
   *NIL
   ITEMS OR STATS TO BE DISPLAYED
   *DOCNO
   *DOCNAME
   *REPEAT

      PRIMARY RECORD (REPEAT)
      *PATIENT
      CONDITIONS FOR RETRIEVAL
      *NIL
      ITEMS OR STATS TO BE DISPLAYED
      *PATNO
      *PATNAME
      *REPEAT

         PRIMARY RECORD (REPEAT)
         *DOCTOR
         CONDITIONS FOR RETRIEVAL
         *NIL
         ITEMS OR STATS TO BE DISPLAYED
         *DOCNO
         *DOCNAME
         *NIL
      *NIL
   *NIL
POSSIBLE PORTS ARE:
(DOCNO)
SELECT ONE OR TYPE NIL  DOCNO
```

Figure 7-5. Query P3: "Display the name and number of a doctor specified at run-time. Also display the name and number of all of his patients, and for each patient display the name and number of all of his doctors."

THE:GOAL: (PROGRAM P3):IS:ATTAINABLE:BY:THE:FOLLOWING:PROGRAM:

```
        PROCEDURE DIVISION.
        PROC1 SECTION.

        PARA-100.
            OPEN AREA A1 A2.
            DISPLAY "DOCNO" "EQ?".
            ACCEPT DOCNO.
            FIND DOCTOR RECORD.
            IF ERRORSTATUS IS NOT EQUAL TO 0  NEXT SENTENCE
                ELSE PERFORM PARA-101.
            CLOSE AREA A1 A2.
            STOP.

        PARA-101.
            GET DOCTOR RECORD.
            DISPLAY DOCNO DOCNAME.
            FIND FIRST TREATMENT RECORD OF TREATING SET.
            MOVE CURRENCY STATUS FOR TREATING SET TO TDBKEY.
            PERFORM PARA-102 UNTIL ERRORSTATUS IS NOT EQUAL TO 0.

        PARA-102.
            MOVE CURRENCY STATUS FOR TREATING SET TO Z2.
            FIND OWNER RECORD OF TREATMENTS SET.
            GET PATIENT RECORD.
            DISPLAY PATNO PATNAME.
            FIND FIRST TREATMENT RECORD OF TREATMENTS SET.
            MOVE CURRENCY STATUS FOR TREATMENTS SET TO TDBKEY.
            PERFORM PARA-103 UNTIL ERRORSTATUS IS NOT EQUAL TO 0.
            MOVE Z2 TO TDBKEY.
            FIND TREATMENT USING TDBKEY.
            FIND NEXT TREATMENT RECORD OF TREATING SET.

        PARA-103.
            MOVE CURRENCY STATUS FOR TREATMENTS SET TO Z1.
            FIND OWNER RECORD OF TREATING SET.
            GET DOCTOR RECORD.
            DISPLAY DOCNO DOCNAME.
            MOVE Z1 TO TDBKEY.
            FIND TREATMENT USING TDBKEY.
            FIND NEXT TREATMENT RECORD OF TREATMENTS SET.
```

Figure 7-6. Program P3.

8.0 COST EFFECTIVENESS

In his paper, Gerritsen [1975] concluded that his DMLP system reduced program generation costs by 77% to 95%. Note that this was in examining costs of executing the DMLP system vs costs of human programming. It did not include costs of generating the DMLP system, however, as Gerritsen pointed out, with savings of such magnitude it should be possible to absorb such costs and still come out ahead.

Figure 8-1 compares the program generation costs of the three programs illustrated in Section 7.0 for the modified DMLP system and the original DMLP system.

Program	Original DMLP			Modified DMLP		
	Run Time	KCS Time	$	Run Time	KCS Time	$
P1	1.37	7151	37.33	1.10	4622	24.78
P2	2.40	10556	55.17	1.24	6318	33.88
P3	1.44	8909	47.08	1.10	5165	27.37

Figure 8-1 Performance of Original DMLP system vs Modified DMLP system.

This data shows a cost reduction of approximately 40%. Of course this is somewhat dependent on the billing algorithm, but cost savings would still result with other algorithms. The cost savings shown in Figure 8-1 could have been further increased by running the modified DMLP with less core. The costs shown are for running both systems with the same core, however, since Micro-Planner is used to a much smaller degree in the modified version, it is possible to run it with approximately 70% of the core requirement of the original system. If

the changeover to LISP discussed in Section 9.2 were implemented still further cost savings could be realized both in execution time and core requirements.

9.0 FURTHER WORK

9.1 Extensions Of The Task Scope

In his thesis, Gerritsen [1975] identified several desired extensions to the scope of programs which the system could generate. These extensions fall mostly into two categories. First, there are extensions to increase program efficiency. These would include such things as eliminating redundant code, making use of ordered sets, ordering disjuncts and conjuncts to qualify or disqualify a given record with minimal testing, and "intelligently" choosing between area and set searchs so as to minimize the number of data base accesses. Second, there are extensions which would increase the scope of possible queries. These would include allowing calculation and storage of temporary items, and conceptually having a "top of the world" record to facilitate such queries as "What is the total number of Doctors?".

Many of the extensions of the first type will require providing additional information in the initial state. While providing the information about ordered sets is no great problem, the information needed to choose between an area search and a set search is not so readily obtainable and would change over time. It would be feasible to have the Data Base Management System update such information. Most extensions of the first type will not involve altering the program generation rules, but will involve altering the planning stage (i.e. HI-IQ's interaction with the user and the its generation of assertions about the query).

AD-A034 390 WHARTON SCHOOL OF FINANCE AND COMMERCE PHILADELPHIA P--ETC F/G 9/2
DETERMINISTIC VERSUS NONDETERMINISTIC PROCEDURE FOR AUTOMATIC P--ETC(U)
OCT 76 D J ROOT N00014-75-C-0462
UNCLASSIFIED 76-10-01 NL

2 OF 2
AD
A034390

END
DATE
FILMED
2-77

9.2 Additional Modifications To The System

As shown in Section 8, the elimination of most of the use of Micro-planner resulted in savings on execution costs of approximately 40%. Since Micro-Planner is now used only to plan the path through the data base and for pattern matching capabilities to access the state description, it should be possible to change the system over completely to LISP and allow it to be run as compiled rather than interpreted functions. This would result in further cost savings.

Besides this changeover to LISP, one additional change is desirable to create a "cleaner" looking system. That is to make the generation of conditional procedures more explicit than it is in the present system. The use of three valued logic (true, false and uncertain) was an effective way of allowing the system to generate conditional procedures when it was operating nondeterministically. However, now that the Program Generator's activities are deterministic, it does not have to be left to the system to discover it needs to generate a conditional procedure through three valued logic. The rules could explicitly show where, when and how conditional procedures are generated, rather than using the less obvious method of uncertain partial preconditions causing the system to generate conditional procedures.

10.0 NONDETERMINISTIC VS DETERMINISTIC PROCEDURES

If the question of the benefits of deterministic vs nondeterministic procedures in automatic program synthesis were to be decided merely on the grounds of the relative costs of program generation, the results in Section 8 would support the assertion that a deterministic route is better. However, the question is not that simple. There are several aspects to the problem, some favor one approach and some favor the other.

1) Given the current state of the art of Artificial Intelligence, if a deterministic method exists for solving a problem, it can usually do so with considerably less machine effort than a nondeterministic method. (This is a conclusion which Wang [1960] drew back in 1960 in relation to Newell and Simon`s [1956, 1957a, 1957b] "Logic Theorist".)

2) Not all problems are currently solveable by deterministic procedures. Futhermore, there is the question of whether all problems will ever be solveable by deterministic procedures.

3) Deterministic procedures are more confined to solving only those problems for which they were designed than are nondeterministic procedures.

As regards this application, for the present the deterministic approach seems best. The task is clearly enough defined that lack of flexibility is not a problem, and plans to expand the system, as mentioned in Section 9, appear to be made easier by having separated the planning stage from the code generation stage. (Note: this separation does not truly require a deterministic approach, Siklossy

and Sikes [1975] are currently working on a system which separates the two functions, but the system currently only works on tasks of fairly low complexity.) Many of the proposed changes will involve only the formulation of the plan as it is set down in the initial state which the Request Handler generates. Therefore in making the necessary changes only one segment of the system will be involved (this is similar to the benefits associated with structured programming).

A major benefit of the deterministic approach is its lower cost. With the extensions outlined in the previous section, the system should be capable of generating a fairly wide range of complex programs for querying a network data base. While other automatic program synthesizers might be able to duplicate the complexity of tasks, the cost would most likely be prohibitive. In the approach used here, the systems actions are almost entirely preplanned, thereby eliminating costs of trial and error. While this limits the systems capabilities to a very specific scope of tasks, it does provide a currently economical system, and the task of preplanning the systems activities will probably help give an understanding of what kind of knowledge a system will have to have to exercise its own "common sense".

Bibliography

Baker, Terry F. and Harlan D. Mills (1973),"Chief Programmer Teams", Datamation, December 1973, pp 58-61.

Buchanan, J. R. (1974),"A Study in Automatic Programming", PhD Thesis, Stanford University.

Buchanan, J. R. and D. C. Luckman (1974),"On Automating the Construction of Programs", Stanford AI Memo, Stanford University.

CODASYL (1971),CODASYL Data Base Task Group April 1971 Report, ACM, New York City.

Date, C. (1976),Introduction to Data Base Systems, Addison-Wesley, Menlo Park, Calif.

Dijkstra, E. W. (1976),A Discipline of Programming, Prentice Hall, Englewood Cliffs, N. J.

Gerritsen, Rob (1974),"Automatically Generated Programs for Information Retrieval; IRP, a Rudimentary System", Carnegie-Mellon Graduate School of Industrial Administration, W.P.-47-73-74.

Gerritsen, Rob (1975),"Understanding Data Structures", PhD Thesis, Carnegie-Mellon University, Pittsburgh, Penn.

Green, Cordell and David Barlow (1975),"Some Rules for the Automatic Synthesis of Programs", Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilsi, Georgia, USSR, 3-8 Sept 1975.

Haseman, William D. and Andrew B. Whinston (1975),"Problem Solving Approach to Data Base Management", Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilsi, Georgia, USSR, 3-8 Sept 1975.

Hewitt, C. (1971),"Description and Theoretical Analysis of Planner", PhD Thesis, Massachusetts Intstitute of Technology.

Hoare, C. A. R. (1969),"An Axiomatic Basis for Computer Programming", CACM 3, October 1969, pp 576-580.

Hoare, C. A. R., O. J. Dahl and E. W. Dijkstra (1972),Structured Programming, Academic Press, New York City.

Hoare, C. A. R. and N. Wirth (1972),"An Axiomatic Definition of the Programming Language PASCAL", Berichte der Fachgruppe Computer - Wissenschaften 6, E.T.H., Zurich, November 1972.

Igarashi, S. R., L. London and D. C. Luckham (1973),"Automatic Program Verification I: A Logical Basis and Implementation", Stanford

AI Memorandum 200, May 1973.

Lee, R. C. T., C. L. Chang and R. J. Waldinger (1974),"An Improved Program-Synthesizing Algorithm and Its Correctness", Communications of the ACM, April 1974 Vol 17 No 4.

Manna, Zohar and Ricahrd Waldinger (1975),"Knowledge and Reasoning in Program Synthesis", Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilsi, Georgia, USSR, 3-8 Sept 1975.

McCarthy, J., P. W. Abrahams, D. J. Edwards, T. P. Hart and M. I. Levin (1972),LISP 1.5 Programmers Manual, MIT Press.

McKeeman, W., J. J. Horning and D. B. Wortman (1970),A Compiler Generator, Prentice Hall, N. J.

Mills, Harlan D. (1975),"The New Math of Computer Programming", Communications of the ACM, January 1975 Vol 18 No 1.

Naur, P. et al (1960),"Report on the Algorithmic Language ALGOL 60", Communications of the ACM No 3 1960.

Newell, A. and H. A. Simon (1956),"The Logic Theory Machine", IRE Transactions on Information Theory 1956.

Newell, A., J. C. Shaw and H. A. Simon (1957a),"Empirical Explorations of the Logic Theory Machine", Proceedings of the Western Joint Computer Conference 1957, pp 218-239.

Newell, A. and J. C. Shaw (1957b),"Programming the Logic Theory Machine", Proceedings of the Western Joint Computer Conference 1957, pp 230-240.

Siklossy, L. and D. A. Sykes (1975),"Automatic Program Sythesis From Example Problems," Advance Papers of the Joint Conference on Artificial Intelligence, Tbilsi, Georgia, USSR, 3-8 Sept 1975.

Sussman, Gerald J. and Drew V. McDermott (1972),"Why Conniving is Better than Planning", Massachusetts Institute of Technology, Artificial Intelligence Memo 255A.

Sussman, Gerald J. and Terry Winograd (1972),"Micro-planner Reference Manual", MIT Project MAC Report.

DISTRIBUTION LIST

Department of the Navy - Office of Naval Research

Data Base Management Systems Project

Defense Documentation Center
Cameron Station
Alexandria, VA 22314

Office of Naval Research
Code 102IP
Arlington, VA 22217

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605

New York Area Office
715 Broadway - 5th Floor
New York, NY 10003

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, DC 20380

Office of Naval Research
Code 458
Arlington, VA 22217

Assistant Chief for Technology
Office of Naval Research
Code 200
Arlington, VA 22217

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, MA 02210

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, DC 20375

Office of Naval Research
Code 455
Arlington, VA 22217

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner
Naval Ship Research and
Development Center
Computation & Mathematics Dept.
Bethesda, MD 20084

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
(OP-9160)
Office of Chief of Naval Operations
Washington, DC 20350

Mr. Kim B. Thompson
Technical Director
Information Systems Division
(OP-91T)
Office of Chief of Naval Operations
Washington, DC 20350

Bureau of Library and
Information Science Research
Rutgers - The State University
189 College Avenue
New Brunswick, NJ 08903
Attn: Dr. Henry Voos

Professor Omar Wing
Columbia University
in the City of New York
Dept. of Electrical Engineering
and Computer Science
New York, NY 10027